

Linear-time string indexing and analysis in small space

Djamal Belazzougui, Fabio Cunial, Juha Kärkkäinen, and Veli Mäkinen

Helsinki Institute for Information Technology

September 22, 2016

Abstract

The field of *succinct data structures* has flourished over the last 16 years. Starting from the compressed suffix array by Grossi and Vitter (STOC 2000) and the FM-index by Ferragina and Manzini (FOCS 2000), a number of generalizations and applications of string indexes based on the Burrows-Wheeler transform (BWT) have been developed, all taking an amount of space that is close to the input size in bits. In many large-scale applications, the construction of the index and its usage need to be considered as one unit of computation. For example, one can compare two genomes by building a common index for their concatenation, and by detecting common substructures by querying the index. Efficient string indexing and analysis in small space lies also at the core of a number of primitives in the data-intensive field of high-throughput DNA sequencing.

We report the following advances in string indexing and analysis. We show that the BWT of a string $T \in \{1, \dots, \sigma\}^n$ can be built in deterministic $O(n)$ time using just $O(n \log \sigma)$ bits of space, where $\sigma \leq n$. Deterministic linear time is achieved by exploiting a new *partial rank* data structure that supports queries in constant time, and that might have independent interest. Within the same time and space budget, we can build an index based on the BWT that allows one to enumerate all the internal nodes of the suffix tree of T . Many fundamental string analysis problems, such as maximal repeats, maximal unique matches, and string kernels, can be mapped to such enumeration, and can thus be solved in deterministic $O(n)$ time and in $O(n \log \sigma)$ bits of space from the input string, by tailoring the enumeration algorithm to some problem-specific computations.

We also show how to build many of the existing indexes based on the BWT, such as the *compressed suffix array*, the *compressed suffix tree*, and the *bidirectional BWT index*, in *randomized* $O(n)$ time and in $O(n \log \sigma)$ bits of space. The previously fastest construction algorithms for BWT, compressed suffix array and compressed suffix tree, which used $O(n \log \sigma)$ bits of space, took $O(n \log \log \sigma)$ time for the first two structures, and $O(n \log^\epsilon n)$ time for the third, where ϵ is any positive constant smaller than one. Contrary to the state of the art, our bidirectional BWT index supports every operation in constant time per element in its output.

This work was partially supported by Academy of Finland under grants 250345 and 284598 (CoECGR).

This work extends results originally presented in ESA 2013 (all authors) and STOC 2014 (Belazzougui).

Author's address: (Helsinki Institute for Information Technology), Department of Computer Science, P.O. Box 68 (Gustaf Hållströmin katu 2b), FIN-00014, University of Helsinki, Finland.

Djamal Belazzougui is currently with the Centre de Reserche sur L'Information Scientifique et Technique, Algeria, and Fabio Cunial is with the Max-Planck Institute of Molecular Cell Biology and Genetics, Germany.

Contents

1	Introduction	3
2	Definitions and preliminaries	6
2.1	Temporary space and working space	6
2.2	Strings	6
2.3	Suffix tree	7
2.4	Rank and select	8
2.5	String indexes	8
3	Building blocks and techniques	11
3.1	Static memory allocation	11
3.2	Batched locate queries	11
3.3	Data structures for prefix-sum queries	12
3.4	Data structures for access, rank, and select queries	12
3.5	Representing the topology of suffix trees	15
3.6	Data structures for monotone minimal perfect hash functions	18
3.7	Data structures for range-minimum and range-distinct queries	19
4	Enumerating all right-maximal substrings	22
5	Building the Burrows-Wheeler transform	31
6	Building string indexes	33
6.1	Building the compressed suffix array	33
6.2	Building BWT indexes	34
6.3	Building the bidirectional BWT index	35
6.4	Building the permuted LCP array	39
6.5	Building the compressed suffix tree	39
7	String analysis	42
7.1	Matching statistics	42
7.2	Maximal repeats, maximal unique matches, maximal exact matches.	43
7.3	String kernels	47

1 Introduction

The suffix tree [67] is a fundamental text indexing data structure that has been used for solving a large number of string processing problems over the last 40 years [2, 32]. The suffix array [46] is another widely popular data structure in text indexing, and although not as versatile as the suffix tree, its space usage is bounded by a smaller constant: specifically, given a string of length n over an alphabet of size σ , a suffix tree occupies $O(n \log n)$ bits of space, while a suffix array takes exactly $n \lceil \log n \rceil$ bits¹.

The last decade has witnessed the rise of *compressed* versions of the suffix array [31, 23] and of the suffix tree [63]. In contrast to their plain versions, they occupy just $O(n \log \sigma)$ bits of space: this shaves a $\Theta(\log_\sigma n)$ factor, thus space becomes just a constant times larger than the original text, which is encoded in exactly $n \log \sigma$ bits. Any operation that can be implemented on a suffix tree (and thus any algorithm or data structure that uses the suffix tree) can be implemented on the *compressed suffix tree* (henceforth denoted by CST) as well, at the price of a slowdown that ranges from $O(1)$ to $O(\log^\epsilon n)$ depending on the operation. Building a CST, however, suffers from a large slowdown if we are restricted to use an amount of space that is only a constant factor away from the space taken by the CST itself. More precisely, a CST can be built in deterministic $O(n \log^\epsilon n)$ time (where ϵ is any constant such that $0 < \epsilon < 1$) and $O(n \log \sigma)$ bits of space [38], or alternatively in deterministic $O(n)$ time and $O(n \log n)$ bits by first employing a linear-time deterministic suffix tree construction algorithm to build the plain suffix tree [21], and then compressing the resulting representation. It can also be built in deterministic $O(n \log \log n)$ time and $O(n \log \sigma \log \log n)$ bits of space (by combining [38] with [37]).

The compressed version of the suffix array (denoted by CSA in what follows) does not suffer from the same slowdown in construction as the compressed suffix tree, since it can be built in deterministic $O(n \log \log \sigma)$ time² and $O(n \log \sigma)$ bits of space [38], or alternatively in deterministic $O(n)$ time and in $O(n \log \sigma \log \log n)$ bits of space [55].

In this paper we show that both the CST and the CSA can be built in *randomized* $O(n)$ time using $O(n \log \sigma)$ bits of space, where randomization comes from the use of *monotone minimal perfect hash functions*³. This seems in contrast to the plain suffix array and suffix tree, which can be built in *deterministic* $O(n)$ time. However, hashing is also necessary to build a representation of the plain suffix tree that supports the fundamental child operation in constant time⁴: building such a plain representation of the suffix tree takes itself randomized $O(n)$ time. If one insists on achieving deterministic linear construction time, then the fastest bound known so far for the child operation is $O(\log \log \sigma)$.

We also show that the key ingredient of compressed text indexes, namely the *Burrows-Wheeler transform* (BWT) of a string [15], can be built in deterministic $O(n)$ time and $O(n \log \sigma)$ bits of space. Such construction rests on the following results, which we believe have independent technical interest and wide applicability to string processing and biological sequence analysis problems. The first result is a data structure that takes at most $n \log \sigma + O(n)$ bits of space, and that supports access and partial rank⁵ queries in constant time, and a related data structure that takes $n \log \sigma (1 + 1/k) + O(n)$ bits of space for any positive integer k , and that supports either access and partial rank queries in constant time and select queries in $O(k)$ time, or select queries in constant time and access and partial rank queries in $O(k)$ time (Lemma 7). Both such data structures can be built in deterministic $O(n)$ time and $o(n)$ bits of space.

In turn, the latter data structure enables an index that takes $n \log \sigma + O(n)$ bits of space, and that allows one to enumerate a rich representation of all the internal nodes of a suffix tree, in overall $O(n)$ time and in $O(\sigma^2 \log^2 n)$ bits of additional space (Lemmas 19 and 22). Such index is our second result of independent interest: we call it the *unidirectional BWT index*. Our enumeration algorithm is easy to implement, to parallelize, and to apply to multiple strings, and it performs a depth-first traversal of the *suffix-link tree*⁶ using a stack that contains at every time at most $\sigma \log n$ nodes. A similar enumeration algorithm, which

¹In this paper $\log n$ stands for $\log_2 n$.

²This bound should actually read as $O(n \cdot \max(1, \log \log \sigma))$.

³Monotone minimal perfect hash functions are defined in Section 3.6.

⁴The constant-time child operation enables e.g. matching a pattern of length m against the suffix tree in $O(m)$ time.

⁵Access, partial rank and select queries are defined in Section 2.

⁶The suffix-link tree is defined in Section 2.3.

performs however a breadth-first traversal of the suffix-link tree, was described in [13]: such algorithm uses a queue that takes $\Theta(n)$ bits of space, and that contains $\Theta(n)$ nodes in the worst case. This number of nodes might be too much for applications that require storing e.g. a real number per node, like weighted string kernels (see [9] and references therein).

We also show that many fundamental operations in string analysis and comparison, with a number of applications to genomics and high-throughput sequencing, can all be performed by enumerating the internal nodes of a suffix tree *regardless of their order*. This allows one to implement all such operations in deterministic $O(n)$ time on top of the unidirectional index, and thus in deterministic $O(n)$ time and $O(n \log \sigma)$ bits of space *directly from the input string*: this is our third result of independent interest. Implementing such string analysis procedures on top of our enumeration algorithm is also practical, as it amounts to few lines of code invoked by a callback function. Using again the enumeration procedure, we give a practical algorithm for building the BWT of the *reverse* of a string given the BWT of the string. Contrary to [53], our algorithm does not need the suffix array and the original string in addition to the BWT.

To build the CST we make use of the *bidirectional BWT index*, a data structure consisting of two BWTs that has a number of applications in high-throughput sequencing [65, 66, 43, 44]. Our fourth result of independent interest consists in showing that, in randomized $O(n)$ time and in $O(n \log \sigma)$ bits of space, we can build a bidirectional BWT index that takes $O(n \log \sigma)$ bits of space and that supports every operation in constant time per element in the output (Theorem 12). This is in contrast to the $O(\sigma)$ or $O(\log \sigma)$ time per element in the output required by existing bidirectional indexes in some key operations. Our fifth result of independent interest is an algorithm that builds the *permuted LCP array* (a key component of the CST), as well as the *matching statistics array*⁷, given a constant-time bidirectional BWT index, in $O(n)$ time and $O(\log n)$ bits of space (Lemmas 31 and 35). Both such algorithms are practical.

The paper consists of a number of other intermediate results, whose logical dependencies are summarized in Figure 1. We suggest to keep this figure at hand in particular while reading Section 6.

⁷The permuted LCP array is defined in Section 2.5. The matching statistics array and related notions are defined in Section 7.

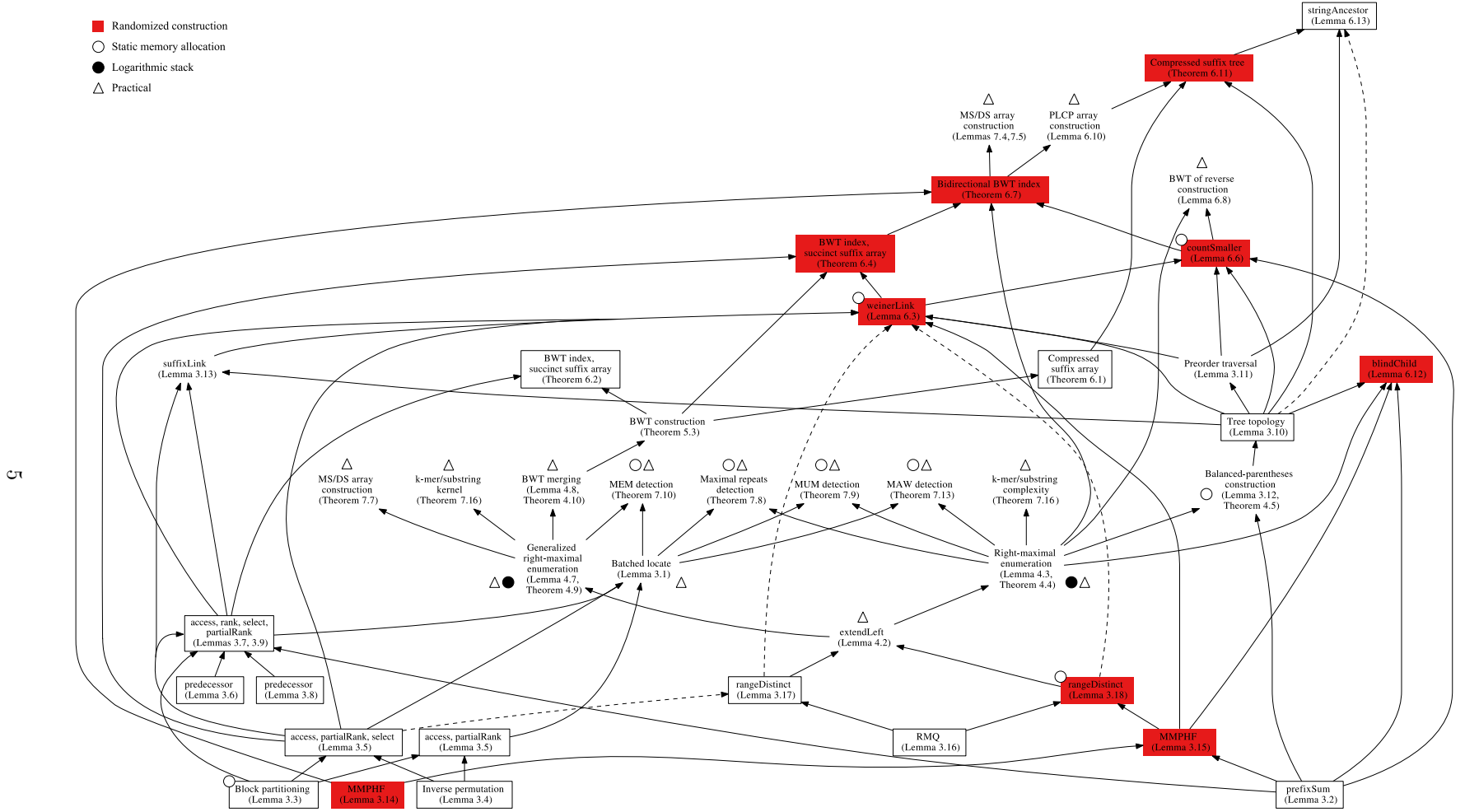


Figure 1: Map of the main data structures (rectangles) and algorithms described in the paper. Data structures whose construction algorithm works in randomized time are highlighted in red. Algorithms that use the static allocation strategy are marked with a white circle (see Section 3.1). Algorithms that use the logarithmic stack technique described in the proof of Lemma 22 are marked with a black circle. Algorithms that are easy to implement in practice are highlighted with a triangle. Arcs indicate logical dependencies. A dashed arc (v, w) means that data structure v is used to build data structure w , but some of the components of v are discarded after the construction of w .

2 Definitions and preliminaries

We work in the RAM model, and we index arrays starting from one. We denote by $i \pmod{1} n$ the function that returns n if $i = 0$, that returns i if $i \in [1..n]$, and that returns 1 if $i = n + 1$.

2.1 Temporary space and working space

We call *temporary space* the size of any region of memory that: (1) is given in input to an algorithm, initialized to a specific state; (2) is read and written (possibly only in part) by the algorithm during its execution; (3) is restored to the original state by the algorithm before it terminates. We call *working space* the maximum amount of memory that an algorithm uses in addition to its input, its output, and its temporary space (if any). The temporary space of an algorithm can be bigger than its working space.

2.2 Strings

A *string* T of length n is a sequence of symbols from the compact alphabet $\Sigma = [1..\sigma]$, i.e. $T \in \Sigma^n$. We assume $\sigma \in o(\sqrt{n}/\log n)$, since for larger alphabets there already exist algorithms for building the data structures described in this paper, in linear time and in $O(n \log \sigma) = O(n \log n)$ bits of working space (for example the linear-time suffix array construction algorithms in [41, 40, 39]). The reason behind our choice of $o(\sqrt{n}/\log n)$ will become apparent in Section 4. We also assume $\#$ to be a separator that does not belong to $[1..\sigma]$, and specifically we set $\# = 0$. In some cases we use multiple distinct separators, denoted by $\#_i = -i + 1$ for integers $i > 0$.

Given a string $T \in [1..\sigma]^n$, we denote by $T[i..j]$ (with i and j in $[1..n]$) a *substring* of T , with the convention that $T[i..j]$ equals the empty string if $i > j$. As customary, we denote by $V \cdot W$ the concatenation of two strings V and W . We call $T[1..i]$ (with $i \in [1..n]$) a *prefix* of T , and $T[j..n]$ (with $j \in [1..n]$) a *suffix* of T .

A *rotation* of T is a string $T[i..n] \cdot T[1..i - 1]$ for $i \in [1..n]$. We denote by $\mathcal{R}(T)$ the set of all *lexicographically distinct* rotations of T . Note that $|\mathcal{R}(T)|$ can be smaller than n , since some rotations of T can be lexicographically identical: this happens if and only if $T = W^k$ for some $W \in [1..\sigma]^+$ and $k > 1$. We are interested only in strings for which all rotations are lexicographically distinct: we often enforce this property by terminating a string with $\#$. We denote by $\mathcal{S}(T)$ the set of all distinct, not necessarily proper, prefixes of rotations of T . In what follows we will use rotations to define a set of notions (like maximal repeats, suffix tree, suffix array, longest common prefix array) that are typically defined in terms of the *suffixes* of a string terminated by $\#$. We do so to highlight the connection between such notions and the *Burrows-Wheeler transform*, one of the key tools used in the following sections, which is defined in terms of rotations. Note that there is a one-to-one correspondence between the i -th rotation of $T\#$ in lexicographic order and the i -th suffix of $T\#$ in lexicographic order.

A *repeat* of T is a string $W \in \mathcal{S}(T)$ such that there are two rotations $T^1 = T[i_1..n] \cdot T[1..i_1 - 1]$ and $T^2 = T[i_2..n] \cdot T[1..i_2 - 1]$, with $i_1 \neq i_2$, such that $T^1[1..|W|] = T^2[1..|W|] = W$. Repeats are substrings of T if $T \in [1..\sigma]^{n-1}\#$. A repeat W is *right-maximal* if $|W| < n$ and there are two rotations $T^1 = T[i_1..n] \cdot T[1..i_1 - 1]$ and $T^2 = T[i_2..n] \cdot T[1..i_2 - 1]$, with $i_1 \neq i_2$, such that $T^1[1..|W|] = T^2[1..|W|] = W$ and $T^1[|W| + 1] \neq T^2[|W| + 1]$. A repeat W is *left-maximal* if $|W| < n$ and there are two rotations $T^1 = T[i_1..n] \cdot T[1..i_1 - 1]$ and $T^2 = T[i_2..n] \cdot T[1..i_2 - 1]$, with $i_1 \neq i_2$, such that $T^1[2..|W| + 1] = T^2[2..|W| + 1] = W$ and $T^1[1] \neq T^2[1]$. Intuitively, a right-maximal (respectively, left-maximal) repeat cannot be extended to the right (respectively, to the left) by a single character, without losing at least one of its occurrences in T . A repeat is *maximal* if it is both left- and right-maximal. If $T \in [1..\sigma]^{n-1}\#$, repeats are substrings of T , and we use the terms left- (respectively, right-) maximal *substring*. Given a string $W \in \mathcal{S}(T)$, we call $\mu(W)$ the number of (not necessarily proper) suffixes of W that are maximal repeats of T , and we set $\mu_T = \max\{\mu(T') : T' \in \mathcal{R}(T)\}$. We say that a repeat W of T is *strongly left-maximal* if there are at least two distinct characters a and b in $[1..\sigma]$ such that both aW and bW are right-maximal repeats of T . Since only a right-maximal repeat W of T can be strongly left-maximal, the set of strongly left-maximal repeats of T is a subset of the maximal repeats of T . Let $W \in \mathcal{S}(T)$, and let $\lambda(W)$ be the number of (not necessarily proper) suffixes of W that are

strongly left-maximal repeats of T . We set $\lambda_T = \max\{\lambda(T') : T' \in \mathcal{R}(T)\}$. Note that $\lambda_T \leq \mu_T$. Other types of repeat will be described in Section 7.

2.3 Suffix tree

Let $\mathcal{T} = \{T^1, T^2, \dots, T^m\}$ be a set of strings on alphabet $[1..\sigma]$. The *trie* of \mathcal{T} is the tree $G = (V, E, \ell)$, with set of nodes V , set of edges E , and labeling function ℓ , defined as follows: (1) every edge $e \in E$ is labeled by exactly one character $\ell(e) \in [1..\sigma]$; (2) the edges that connect a node to its children have distinct labels; (3) the children of a node are sorted lexicographically according to the labels of the corresponding edges; (4) there is a one-to-one correspondence between V and the set of distinct prefixes of strings in \mathcal{T} . Note that, if no string in \mathcal{T} is a prefix of another string in \mathcal{T} , there is a one-to-one correspondence between the elements of \mathcal{T} and the leaves of the trie of \mathcal{T} .

Given a trie, we call *unary path* a maximal sequence v_1, v_2, \dots, v_k such that $v_i \in V$ and v_i has exactly one child, for all $i \in [1..k]$. By *collapsing a unary path* we mean transforming $G = (V, E, \ell)$ into a tree $G' = (V' \setminus \{v_1, \dots, v_k\}, (E \setminus \{(v_0, v_1), (v_1, v_2), \dots, (v_k, v_{k+1})\}) \cup \{(v_0, v_{k+1})\}, \ell')$, where v_0 is the parent of v_1 in G , v_{k+1} is the only child of v_k in G , $\ell'(e) = \ell(e)$ for all $e \in E \cap E'$, and $\ell'((v_0, v_{k+1}))$ is the concatenation $\ell(v_0, v_1) \cdot \ell(v_1, v_2) \cdot \dots \cdot \ell(v_k, v_{k+1})$. Note that ℓ' labels the edges of G' with strings rather than with single characters. Given a trie, we call *compact trie* the labeled tree obtained by collapsing all unary paths in the trie. Every node of a compact trie has either zero or at least two children.

Definition 1 ([67]). Let $T \in [1..\sigma]^n$ be a string such that $|\mathcal{R}(T)| = n$. The *suffix tree* $\text{ST}_T = (V, E, \ell)$ of T is the compact trie of $\mathcal{R}(T)$.

Note that ST_T is not defined if some rotations of T are lexicographically identical, and that there is a one-to-one correspondence between the leaves of the suffix tree of T and the elements of $\mathcal{R}(T)$. Since the suffix tree of T has precisely n leaves, and since every internal node is branching, there are at most $n - 1$ internal nodes. We denote by $\text{sp}(v)$, $\text{ep}(v)$, and $\text{range}(v)$ the left-most leaf, the right-most leaf, and the set of all leaves in the subtree of an internal node v , respectively. We denote by $\ell(e)$ the label of an edge $e \in E$, and by $\ell(v)$ the string $\ell(r, v_1) \cdot \ell(v_1, v_2) \cdot \dots \cdot \ell(v_{k-1}, v)$, where $r \in V$ is the root of the tree, and $r, v_1, v_2, \dots, v_{k-1}, v$ is the path of $v \in V$ in the tree. We say that node v has *string depth* $|\ell(v)|$. We call w the *proper locus* of string W if the search for W starting from the root of ST_T ends at an edge $(v, w) \in E$. Note that there is a one-to-one correspondence between the set of internal nodes of ST_T and the set of right-maximal repeats of T . Moreover, the set of all left-maximal repeats of T enjoys the *prefix closure* property, in the sense that if a repeat is left-maximal, so is any of its prefixes. It follows that the maximal repeats of T form an induced subgraph of the suffix tree of T , rooted at r .

Given strings T^1, T^2, \dots, T^m with $T^i \in [1..\sigma]^{n_i}$ for $i \in [1..m]$, assume that $|\mathcal{R}(T^i)| = n_i$ for all $i \in [1..m]$, and that $\mathcal{R}(T^i) \cap \mathcal{R}(T^j) = \emptyset$ for all $i \neq j$ in $[1..m]$. We call *generalized suffix tree* the compact trie of $\mathcal{R}(T^1) \cup \mathcal{R}(T^2) \cup \dots \cup \mathcal{R}(T^m)$. Note that, if string W labels an internal node of the suffix tree of a string T^i , then it also labels an internal node of the generalized suffix tree. However, there could be an internal node v in the generalized suffix tree $G = (V, E, \ell)$ such that $\ell(v)$ does not label an internal node in any T^i . This means that: (1) if $\ell(v) \in \mathcal{S}(T^i)$, then it is always followed by the same character a_i in every rotation of T^i ; (2) there are at least two strings T^i and T^j , with $i \neq j$, such that $a_i \neq a_j$. A node v in the generalized suffix tree could be such that all leaves in the subtree rooted at v are rotations of the same string T^i : we call such a node *pure*, and we call it *impure* otherwise.

Let the label $\ell(v)$ of an internal node v of $\text{ST}_T = (V, E, \ell)$ be aW , with $a \in \Sigma$ and $W \in \Sigma^*$. Since W occurs at all positions where aW occurs, there must be a node $w \in V$ with $\ell(w) = W$, otherwise v would not be a node of the suffix tree. We say that there is a *suffix link from v to w labelled by a* , and we write $\text{suffixLink}(v) = w$. More generally, we say that the set of labels of internal nodes of ST_T enjoys the *suffix closure* property, in the sense that if a string W belongs to this set, so does every one of its suffixes. If $T \in [1..\sigma]^{n-1}\#$, we define $\text{suffixLink}(v)$ for leaves v of ST_T as well: the suffix link from a leaf leads either to another leaf, or to the root of ST_T . The graph that consists of the set of internal nodes of ST_T and of the set of suffix links, is a trie rooted at the same root node as ST_T : we call such trie the *suffix-link tree* SLT_T of T . Note that the suffix-link tree might contain unary paths, and that traversing the suffix-link tree

allows one to enumerate all nodes of the suffix tree. Note also that extending to the left a repeat that is not right-maximal does not lead to a right-maximal repeat. We exploit this property in Section 4 to enumerate all nodes of the suffix tree in small space, storing neither the suffix tree nor the suffix-link tree explicitly. Note that every leaf of the suffix-link tree has more than one Weiner link, or its label has length $n - 1$. Thus, the set of all maximal repeats of T coincides with the set of all the internal nodes of the suffix-link tree with at least two (implicit or explicit) Weiner links, and with a subset of all the leaves of the suffix-link tree.

Inverting the direction of all suffix links yields the so-called *explicit Weiner links*. Given a node $v \in V$ and a character $a \in \Sigma$, it might happen that string $a\ell(v) \in \mathcal{S}(T)$, but that it does not label any internal node of ST_T : we call all such extensions of internal nodes *implicit Weiner links*. An internal node might have multiple outgoing Weiner links (possibly both explicit and implicit), and all such Weiner links have distinct labels. The constructions described in this paper rest on the fact that the total number of explicit and implicit Weiner links is small:

Observation 1. *Let $T \in [1..\sigma]^n$ be a string such that $|\mathcal{R}(T)| = n$. The number of suffix links, explicit Weiner links, and implicit Weiner links in the suffix tree of T are upper bounded by $n - 2$, $n - 2$, and $3n - 3$, respectively.*

Proof. Each of the at most $n - 2$ internal nodes of the suffix tree (other than the root) has a suffix link. Each explicit Weiner link is the inverse of a suffix link, so their total number is also at most $n - 2$.

Consider an internal node v with only one implicit Weiner link $e = (\ell(v), a\ell(v))$. The number of such nodes, and thus the number of such implicit Weiner links, is bounded by $n - 1$. Call these the implicit Weiner links of type I, and the remaining the implicit Weiner links of type II. Consider an internal node v with two or more implicit Weiner links, and let Σ_v be the set of labels of all Weiner links from v . Since $|\Sigma_v| > 1$, there is an internal node w in the suffix tree $\text{ST}_{\underline{T}}$ of the *reverse* \underline{T} of T , labeled by the reverse $\ell(v)$ of $\ell(v)$: every $c \in \Sigma_v$ can be mapped to a distinct edge of $\text{ST}_{\underline{T}}$ connecting w to one of its children. This is an injective mapping from all type II implicit Weiner links to the at most $2n - 2$ edges of the suffix tree of \underline{T} . The sum of type I and type II Weiner links, i.e. the number of all implicit Weiner links, is hence bounded by $3n - 3$. \square

Slightly more involved arguments push the upper bound on the number of implicit Weiner links down to n .

2.4 Rank and select

Given a string $S \in [1..\sigma]^n$, we denote by $\text{rank}_c(S, i)$ the number of occurrences of character $c \in [1..\sigma]$ in $S[1..i]$, and we denote by $\text{select}_c(S, j)$ the position i of the j -th occurrence of c in S , i.e. $j = \text{rank}_c(S, \text{select}_c(S, j))$. We use $\text{partialRank}(S, i)$ as a shorthand for $\text{rank}_{S[i]}(S, i)$. Data structures to support such operations efficiently will be described in the sequel. Here we just recall that it is possible to represent a bitvector of length n using $n + o(n)$ bits of space, such that rank and select queries can be supported in constant time (see e.g. [16, 47]). This representation can be built in $O(n)$ time and in $o(n)$ bits of working space. Rank and select data structures can be used to implement a *representation* of a string S that supports operation $\text{access}(S, i) = S[i]$ without storing S itself.

2.5 String indexes

Sorting the set of rotations of a string yields an index that can be used for supporting pattern matching by binary search:

Definition 2 ([46]). *Let $T \in [1..\sigma]^n$ be a string such that $|\mathcal{R}(T)| = n$. The suffix array $\text{SA}_T[1..n]$ of T is the permutation of $[1..n]$ such that $\text{SA}_T[i] = j$ iff rotation $T[j..n] \cdot T[1..j - 1]$ has rank i in the list of all rotations of T taken in lexicographic order.*

Note that SA_T is not defined if some rotations of T are lexicographically identical. We denote by $\text{range}(W) = [\text{sp}(W)..\text{ep}(W)]$ the maximal interval of SA_T whose rotations are prefixed by W . Note that

$\text{range}(W)$, $\text{sp}(W)$ and $\text{ep}(W)$ are in one-to-one correspondence with $\text{range}(v)$, $\text{sp}(v)$ and $\text{ep}(v)$ of a node v of the suffix tree of T such that $\ell(v) = W$. We will often use such notions interchangeably.

The *longest common prefix array* stores the length of the longest common prefix between every two consecutive rotations in the suffix array:

Definition 3 ([46]). Let $T \in [1..\sigma]^n$ be a string such that $|\mathcal{R}(T)| = n$, and let $p(i, j)$ be the function that returns the longest common prefix between the rotation that starts at position $\text{SA}_T[i]$ in T and the rotation that starts at position $\text{SA}_T[j]$ in T . The longest common prefix array of T , denoted by $\text{LCP}_T[1..n]$, is defined as follows: $\text{LCP}_T[1] = 0$ and $\text{LCP}_T[i] = p(i, i-1)$ for all $i \in [2..n]$. The permuted longest common prefix array of T , denoted by $\text{PLCP}_T[1..n]$, is the permutation of LCP_T in string order, i.e. $\text{PLCP}_T[\text{SA}_T[i]] = \text{LCP}_T[i]$ for all $i \in [1..n]$.

The main tool that we use in this paper for obtaining space-efficient index structures is a permutation of T induced by its suffix array:

Definition 4 ([15]). Let $T \in [1..\sigma]^n$ be a string such that $|\mathcal{R}(T)| = n$. The Burrows-Wheeler transform of T , denoted by BWT_T , is the permutation $L[1..n]$ of T such that $L[i] = T[\text{SA}_T[i] - 1 \pmod{n}]$ for all $i \in [1..n]$.

Like SA_T , BWT_T cannot be uniquely defined if some rotations of T are lexicographically identical. Given two strings S and T such that $\mathcal{R}(S) \cap \mathcal{R}(T) = \emptyset$, we say that the BWT of $\mathcal{R}(S) \cup \mathcal{R}(T)$ is the string obtained by sorting $\mathcal{R}(S) \cup \mathcal{R}(T)$ lexicographically, and by printing the character that precedes the starting position of each rotation. Note that either $\mathcal{R}(S) \cap \mathcal{R}(T) = \emptyset$, or $\mathcal{R}(S) = \mathcal{R}(T)$.

A key feature of the BWT is that it is *reversible*: given $\text{BWT}_T = L$, one can reconstruct the unique T of which L is the Burrows-Wheeler transform. Indeed, let V and W be two rotations of T such that V is lexicographically smaller than W , and assume that both V and W are preceded by character a in T . It follows that rotation aV is lexicographically smaller than rotation aW , thus there is a bijection between rotations *preceded by* a and rotations that *start with* a that preserves the relative order among such rotations. Consider thus the rotation that starts at position i in T , and assume that it corresponds to position p_i in SA_T (i.e. $\text{SA}_T[p_i] = i$). If $L[p_i] = a$ is the k -th occurrence of a in L , then the rotation that starts at position $i-1$ in T must be the k -th rotation that starts with a in SA_T , and its position p_{i-1} in SA_T must belong to the compact interval $\text{range}(a)$ that contains all rotations that start with a . For historical reasons, the function that projects the position p_i in SA_T of a rotation that starts at position i , to the position p_{i-1} in SA_T of the rotation that starts at position $i-1 \pmod{n}$, is called **LF** (or *last-to-first mapping* [22, 23], and it is defined as $\text{LF}(i) = j$, where $\text{SA}[j] = \text{SA}[i] - 1 \pmod{n}$. Note that reconstructing T from its BWT requires to know the starting position in T of its lexicographically smallest rotation.

Let again L be the Burrows-Wheeler transform of a string $T \in [1..\sigma]^n$, and assume that we have an array $C[1..\sigma]$ that stores in $C[c]$ the number of occurrences in T of all characters strictly smaller than c , that is the sum of the frequency of all characters in $[1..c-1]$. Note that $C[1] = 0$, and that $C[c] + 1$ is the position in SA_T of the first rotation that starts with character c . It follows that $\text{LF}(i) = C[L[i]] + \text{rank}_{L[i]}(L, i)$.

Function **LF** can be extended to a *backward search* algorithm which counts the number of occurrences in T of a string W , in $O(|W|)$ steps, considering iteratively suffixes $W[i..|W|]$ with i that goes from $|W|$ to one [22, 23]. Given the interval $[i_1..j_1]$ that corresponds to a string V and a character c , the interval $[i_2..j_2]$ that corresponds to string cV can be computed as $i_2 = \text{rank}_c(i_1 - 1) + C[c] + 1$ and $j_2 = \text{rank}_c(j_1) + C[c]$. If $i_2 > j_2$, then $cV \notin \mathcal{S}(T)$. Note that, if W is a right-maximal repeat of T , a step of backward search corresponds to taking an explicit or implicit Weiner link in ST_T . The time for computing a backward step is dominated by the time needed to perform a **rank** query, which is typically $O(\log \log \sigma)$ [27] or $O(\log \sigma)$ [29].

The inverse of function **LF** is called ψ for historical reasons [28, 60], and it is defined as follows. Assume that position i in SA_T corresponds to rotation aW with $a \in [1..\sigma]$: since a satisfies $C[a] < i \leq C[a+1]$, it can be computed from i by performing $\text{select}_0(C', i) - i + 1$ on a bitvector C' that represents C with $\sigma - 1$ ones and n zeros, and that is built as follows: we append $C[i+1] - C[i]$ zeros followed by a one for all $i \in [1..\sigma - 1]$, and we append $n - C[\sigma]$ zeros at the end. Function $\psi(i)$ returns the lexicographic rank of rotation W , given the lexicographic rank i of rotation aW , as follows: $\psi(i) = \text{select}_a(\text{BWT}_T, i - C[a])$.

Combining the BWT and the C array gives rise to the following index, which is known as *FM-index* in the literature [22, 23]:

Definition 5. Given a string $T \in [1..\sigma]^n$, a BWT index on T is a data structure that consists of:

- $\text{BWT}_{T\#}$, with support for rank (and select) queries;
- the integer array $C[0..\sigma]$, that stores in $C[c]$ the number of occurrences in $T\#$ of all characters strictly smaller than c .

The following lemma derives immediately from function LF:

Lemma 1. Given the BWT index of a string $T \in [1..\sigma]^{n-1}\#$, there is an algorithm that outputs the sequence $\text{SA}_T^{-1}[n], \text{SA}_T^{-1}[n-1], \dots, \text{SA}_T^{-1}[1]$, in $O(t)$ time per value in the output, in $O(nt)$ total time, and in $O(\log n)$ bits of working space, where t is the time for performing function LF.

So far we have only described how to support *counting* queries, and we are still not able to *locate* the starting positions of a pattern P in string T . One way of doing this is to *sample* suffix array values, and to extract the missing values using the LF mapping. Adjusting the sampling rate r gives different space/time tradeoffs. Specifically, we sample all the values of $\text{SA}_{T\#}[i]$ that satisfy $\text{SA}_{T\#}[i] = 1 + rk$ for $0 \leq k < n/r$, and we store such samples consecutively, in the same order as in $\text{SA}_{T\#}$, in array **samples** $[1..\lceil n/r \rceil]$. Note that this is equivalent to sampling every r positions in *string order*. We also mark in a bitvector $B[1..n]$ the positions of the suffix array that have been sampled, that is we set $B[i] = 1$ if $\text{SA}_{T\#}[i] = 1 + rk$, and we set $B[i] = 0$ otherwise. Combined with the LF mapping, this allows one to compute $\text{SA}_{T\#}[i]$ in $O(rt)$ time, where t is the time required for function LF. One can set $r = \log^{1+\epsilon} n / \log \sigma$ for any given $\epsilon > 0$ to have the samples fit in $(n/r) \log n = n \log \sigma / \log^\epsilon n = o(n \log \sigma)$ bits, which is asymptotically the same as the space required for supporting counting queries. This setting implies that the extraction of $\text{SA}_{T\#}[i]$ takes $O(\log^{1+\epsilon} nt / \log \sigma)$ time. The resulting collection of data structures is called *succinct suffix array* (see e.g. [51]).

Succinct suffix arrays can be further extended into *self-indexes*. A self-index is a succinct representation of a string T that, in addition to supporting count and locate queries on arbitrary strings provided in input, allows one to access any substring of T by specifying its starting and ending position. In other words, a self-index for T completely replaces the original string T , which can be discarded. Recall that we can reconstruct the whole string $T\#$ from $\text{BWT}_{T\#}$ by applying function LF iteratively. To reconstruct arbitrary substrings efficiently, it suffices to store, for every sampled position $1 + ri$ in string $T\#$, the position of suffix $T[1 + ri..n]$ in $\text{SA}_{T\#}$: specifically, we use an additional array **pos2rank** $[1..\lceil n/r \rceil]$ such that **pos2rank** $[i] = j$ if $\text{SA}_{T\#}[j] = 1 + ri$ [23]. Note that **pos2rank** can be seen itself as a sampling of the *inverse suffix array* at positions $1 + ri$, and that it takes the same amount of space as array **samples**. Given an interval $[e..f]$ in string $T\#$, we can use **pos2rank** $[k]$ to go to the position i of suffix $T[1 + rk..n]$ in $\text{SA}_{T\#}$, where $k = \lceil (f-1)/r \rceil$ and $1 + rk$ is the smallest sampled position greater than or equal to f in $T\#$. We can then apply LF mapping $1 + rk - e$ times starting from i : the result is the whole substring $T[e..1 + rk]$ printed from right to left, thus we can return its proper prefix $T[e..f]$. The running time of this procedure is $O((f - e + r)t)$.

Making a succinct suffix array a self-index does not increase its asymptotic space complexity. We can thus define the succinct suffix array as follows:

Definition 6. Given a string $T \in [1..\sigma]^n$, the succinct suffix array of T is a data structure that takes $n \log \sigma (1 + o(1)) + O((n/r) \log n)$ bits of space, where r is the sampling rate, and that supports the following queries:

- **count**(P): returns the number of occurrences of string $P \in [1..\sigma]^m$ in T .
- **locate**(i): returns $\text{SA}_{T\#}[i]$.
- **substring**(e, f): returns $T[e..f]$.

The following result is an immediate consequence of Lemma 1:

Lemma 2. *The succinct suffix array of a string $T \in [1..\sigma]^n$ can be built from the BWT index of T in $O(nt)$ time and in $O(\log n)$ bits of working space, where t is the time for performing function LF.*

In Section 6 we define additional string indexes used in this paper, like the *compressed suffix array*, the *compressed suffix tree*, and the *bidirectional BWT index*.

3 Building blocks and techniques

3.1 Static memory allocation

Let \mathcal{A} be an algorithm that builds a set of arrays by iteratively appending new elements to their end. In all cases described in this paper, the final size of all growing arrays built by \mathcal{A} can be precomputed by running a slightly modified version \mathcal{A}' of \mathcal{A} that has the same time and space complexity as \mathcal{A} . Thus, we always restructure \mathcal{A} as follows: first, we run \mathcal{A}' to precompute the final size of all growing arrays built by \mathcal{A} ; then, we allocate a single, contiguous region of memory that is large enough to contain all the arrays built by \mathcal{A} , and we compute the starting position of each array inside the region; finally, we run \mathcal{A} using such positions. This strategy avoids memory fragmentation in practice, and in some cases, for example in Section 3.5, it even allows us to achieve better space bounds. See Figure 1 for a list of all algorithms in the paper that use this technique.

3.2 Batched locate queries

In this paper we will repeatedly need to resolve a *batch* of queries $\text{locate}(i) = \text{SA}_{T\#}[i]$ issued on a set of distinct values of i in $[1..n]$, where $T \in [1..\sigma]^{n-1}$. The following lemma describes how to answer such queries using just the BWT of T and a data structure that supports function LF:

Lemma 3. *Let $T \in [1..\sigma]^{n-1}$ be a string. Given the BWT of $T\#$, a data structure that supports function LF, and a list `pairs[1..occ]` of pairs (i_k, p_k) , where $i_k \in [1..n]$ is a position in $\text{SA}_{T\#}$ and p_k is an integer for all $k \in [1..\text{occ}]$, we can transform every pair (i_k, p_k) in `pairs` into the corresponding pair $(\text{SA}_{T\#}[i_k], p_k)$, possibly altering the order of list `pairs`, in $O(nt + \text{occ})$ time and in $O(\text{occ} \cdot \log n)$ bits of working space, where t is the time taken to perform function LF.*

Proof. Assume that we could use a bitvector `marked[1..n]` such that `marked` $[i_k] = 1$ for all the distinct i_k that appear in `pairs`. Building `marked` from `pairs` takes $O(n + \text{occ})$ time. Then, we invert $\text{BWT}_{T\#}$ in $O(nt)$ time. During this process, whenever we are at a position i in $\text{BWT}_{T\#}$, we also know the corresponding position $\text{SA}_{T\#}[i]$ in $T\#$: if `marked` $[i] = 1$, we append pair $(i, \text{SA}_{T\#}[i])$ to a temporary array `translate[1..occ]`. At the end of this process, the pairs in `translate` are in reverse string order: thus, we sort both `translate` and `pairs` in suffix array order. Finally, we perform a linear, simultaneous scan of the two sorted arrays, replacing (i_k, p_k) in `pairs` with $(\text{SA}_{T\#}[i_k], p_k)$ using the corresponding pair $(i_k, \text{SA}_{T\#}[i_k])$ in `translate`.

If $\text{occ} \geq n/\log n$, `marked` fits in $O(\text{occ} \cdot \log n)$ bits. Otherwise, rather than storing `marked[1..n]`, we use a smaller bitvector `marked'[1..n/h]` in which we set `marked'` $[i] = 1$ iff there is an $i_k \in [hi..h(i+1)-1]$. As we invert $\text{BWT}_{T\#}$, we check whether the block i/h that contains the current position i in the BWT, is such that `marked'` $[i/h] = 1$. If this is the case, we binary search i in `pairs`. Every such binary search takes $O(\log \text{occ})$ time, and we perform at most $h \cdot \text{occ}$ binary searches in total. Setting $h = n/(\text{occ} \cdot \log n)$ makes `marked'` fit in $\text{occ} \cdot \log n$ bits, and it makes the total time spent in binary searches $O(n/(\log n/\log \text{occ})) \in O(n)$.

Now if $\text{occ} \geq \sqrt{\log n}$, we sort array `pairs[1..occ]` using radix sort: specifically, we interpret each pair (i_k, p_k) as a triple $(\text{msb}(i_k), \text{lsb}(i_k), p_k)$, where $\text{msb}(x)$ is a function that returns the most significant $\lceil (\log n)/2 \rceil$ bits of x and $\text{lsb}(x)$ is a function that returns the least significant $\lfloor (\log n)/2 \rfloor$ bits of x . Since the resulting primary and secondary keys belong to the range $[1..2\sqrt{n}]$, sorting both `pairs` and `translate` takes $O(\sqrt{n} + \text{occ})$ time and $O((\sqrt{n} + \text{occ}) \log n) \in O(\text{occ} \log n)$ bits of working space. If $\text{occ} < \sqrt{\log n}$ we just sort array `pairs[1..occ]` using standard comparison sort, in time $O(\text{occ} \log n) \in O(\sqrt{\log n} \log n)$. \square

3.3 Data structures for prefix-sum queries

A *prefix-sum data structure* supports the following query on an array of numbers $A[1..n]$: given $i \in [1..n]$, return $\sum_{j=1}^i A[j]$. The following well-known result, which we will use extensively in what follows, derives from combining Elias-Fano coding [18, 20] with bitvectors indexed to support the **select** operation in constant time:

Lemma 4 ([54]). *Given a representation of an array of integers $A[1..n]$ whose total sum is U , that allows one to access its entries from left to right, we can build in $O(n)$ time and in $O(\log U)$ bits of working space a data structure that takes $n(2 + \lceil \log(U/n) \rceil) + o(n)$ bits of space and that answers prefix-sum queries in constant time.*

3.4 Data structures for access, rank, and select queries

We conceptually split a string S of length n into $N = \lceil n/\sigma \rceil$ blocks of size σ each, except possibly for the last block which might be smaller. Specifically, block number $i \in [1..N-1]$, denoted by S^i , covers substring $S[\sigma(i-1)+1..\sigma i]$, and the last block S^N covers substring $S[\sigma(N-1)+1..n]$. The purpose of splitting S into blocks consists in translating global operations on S into local operations on a block: for example, **access**(i) can be implemented by issuing **access**($i - \sigma(b-1)$) on block $b = \lceil i/\sigma \rceil$. The construction we describe in this section largely overlaps with [27].

We use $f(c)$ to denote the frequency of character c in S , $f(c, b)$ to denote the frequency of character c in S^b , and $C^b[c]$ as a shorthand for $\sum_{a=1}^{c-1} f(a, b)$. We encode the block structure of S using bitvector $\mathbf{freq} = \mathbf{freq}_1 \mathbf{freq}_2 \dots \mathbf{freq}_\sigma$, where bitvector $\mathbf{freq}_c[1..f(c) + N]$ is defined as follows:

$$\mathbf{freq}_c = 10^{f(c,1)} 10^{f(c,2)} 1 \dots 10^{f(c,N)}$$

Note that \mathbf{freq} takes at most $2n + \sigma - 1$ bits of space: indeed, every \mathbf{freq}_c contains exactly N ones, thus the total number of ones in all bitvectors is $\sigma \lceil n/\sigma \rceil \leq n + \sigma - 1$, and the total number of zeros in all bitvectors is $\sum_{c \in [1..\sigma]} f(c) = n$. Note also that a **rank** or **select** operation on a specific \mathbf{freq}_c can be translated in constant time into a **rank** or **select** operation on \mathbf{freq} . Bitvector \mathbf{freq} can be computed efficiently:

Lemma 5. *Given a string $S \in [1..\sigma]^n$, vector \mathbf{freq} can be built in $O(n)$ time and in $o(n)$ bits of working space.*

Proof. We use the static allocation strategy described in Section 3.1: specifically, we first compute $f(c)$ for all $c \in [1..\sigma]$ by scanning S and incrementing corresponding counters. Then, we compute the size of each bitvector \mathbf{freq}_c and we allocate a contiguous region of memory for \mathbf{freq} . Storing all $f(c)$ counters takes $\sigma \log n \leq (\sqrt{n}/\log n) \log n = o(n)$ bits of space. Finally, we scan S once again: whenever we see the beginning of a new block, we append a one to the end of every \mathbf{freq}_c , and whenever we see an occurrence of character c , we append a zero to the end of \mathbf{freq}_c . The total time taken by this process is $O(n)$, and the pointers to the current end of each \mathbf{freq}_c in \mathbf{freq} take $o(n)$ bits of space overall. \square

Vector \mathbf{freq}_c , indexed to support **rank** or **select** operations in constant time, is all we need to translate in constant time a **rank** or **select** operation on S into a corresponding operation on a block of S : thus, we focus just on supporting **rank** and **select** operations *inside a block of S* in what follows.

For this purpose, let X^b be the string $1^{f(1,b)} 2^{f(2,b)} \dots \sigma^{f(\sigma,b)}$. S^b can be seen as a permutation of X^b : let $\pi_b : [1..\sigma] \mapsto [1..\sigma]$ be the function that maps a position in S^b onto a position in X^b , and let $\pi_b^{-1} : [1..\sigma] \mapsto [1..\sigma]$ be the function that maps a position in X^b to a position in S^b . A possible choice for such permutation functions is:

$$\pi_b(i) = C^b[S^b[i]] + \mathbf{rank}_{S^b}(S^b[i], i) \quad (1)$$

$$\pi_b^{-1}(i) = \mathbf{select}_{S^b}(i - C^b[c], c) \quad (2)$$

where $c = X^b[i]$ is the only character that satisfies $C^b[c] < i \leq C^b[c+1]$. We store explicitly just one of π_b and π_b^{-1} , so that random access to any element of the stored permutation takes constant time, and we represent the other permutation *implicitly*, as described in the following lemma:

Lemma 6 ([48]). *Given a permutation $\pi[1..n]$ of sequence $1, 2, \dots, n$, there is a data structure that takes $(n/k) \log n + n + o(n)$ bits of space in addition to π itself, and that supports query $\pi^{-1}[i]$ for any $i \in [1..n]$ in $O(k)$ time, for any integer $k \geq 1$. This data structure can be built in $O(n)$ time and in $o(n)$ bits of working space. The query and the construction algorithms assume constant time access to any element $\pi[i]$.*

Proof. A permutation $\pi[1..n]$ of sequence $1, 2, \dots, n$ can be seen as a collection of *cycles*, where the number of such cycles ranges between one and n . Indeed, consider the following iterated version of the permutation operator:

$$\pi^t[i] = \begin{cases} i & \text{if } t = 0 \\ \pi[\pi^{t-1}[i]] & \text{if } t > 0 \end{cases}$$

We say that a position i in π belongs to a cycle of length t , where t is the smallest positive integer such that $\pi^t[i] = i$. Note that π can be decomposed into cycles in linear time and using n bits of working space, by iterating operator π from position one, by marking in a bitvector all the positions that have been touched by such iteration, and by repeating the process from the next position in π that has not been marked.

If a cycle contains a number of arcs t greater than a predefined threshold k , it can be subdivided into $\lceil t/k \rceil$ paths containing at most k arcs each. We store in a dictionary the first vertex of each path, and we associate with it a pointer to the first vertex of the path that *precedes* it. That is, given a cycle $x, \pi[x], \pi^2[x], \dots, \pi^{t-1}[x], x$, the dictionary stores the set of pairs:

$$\left\{ (x, \pi^{k(\lceil t/k \rceil - 1)}[x]) \right\} \cup \left\{ (\pi^{ik}[x], \pi^{(i-1)k}[x]) : i \in [1..(\lceil t/k \rceil - 1)] \right\}.$$

Then, we can determine $\pi^{-1}[i]$ for any value i in $O(k)$ time, by successively computing $i, \pi[i], \pi^2[i], \dots, \pi^k[i]$ and by querying the dictionary for every vertex in such sequence. As soon as the query is successful for some $\pi^j[i]$ with $j \in [0..k]$, we get $\pi^{j-k}[i]$ from the dictionary and we compute the sequence $\pi^{j-k}[i], \pi^{j-k+1}[i], \pi^{j-k+2}[i], \dots, \pi^{-1}[i], i$, returning $\pi^{-1}[i]$. The dictionary can be implemented using a table and a bitvector of size n with **rank** support, which marks the first element of each path of length k of each cycle. \square

Combining Lemma 5 and Lemma 6 with Equations 1 and 2, we obtain the key result of this section:

Lemma 7. *Given a string of length n over alphabet $[1..\sigma]$, we can build the following data structures in $O(n)$ time and in $o(n)$ bits of working space:*

- *a data structure that takes at most $n \log \sigma + 4n + o(n)$ bits of space, and that supports **access** and **partialRank** in constant time;*
- *a data structure that takes $n \log \sigma (1 + 1/k) + 5n + o(n)$ bits of space for any positive integer k , and that supports either **access** and **partialRank** in constant time and **select** in $O(k)$ time, or **select** in constant time and **access** and **partialRank** in $O(k)$ time.*

*Neither of these data structures requires the original string to support **access**, **partialRank** and **select**.*

Proof. In addition to the data structures built in Lemma 5, we store π_b explicitly for every S^b , spending overall $n \log \sigma$ bits of space. Note that π_b can be computed from S^b in linear time for all $b \in [1..N]$. We also store C^b for every S^b as a bitvector of 2σ bits that coincides with a unary encoding of X^b (that is we store $10^{f(1,b)}10^{f(2,b)} \dots 10^{f(\sigma,b)}$): given a position i in block S^b , we can determine the character c that satisfies $C^b[c] < \pi_b[i] \leq C^b[c+1]$ using a **select** and a **rank** query on such bitvector, thus implementing **access** to $S^b[i]$ in constant time. In turn, this allows one to implement **partialRank** _{S^b} (i) in constant time using Equation 1.

A **select** query on C^b , combined with the implicit representation of π_b^{-1} described in Lemma 6, allows one to implement **select** on S^b in $O(k)$ time, at the cost of $(\sigma/k) \log \sigma + \sigma + o(\sigma)$ bits of additional space per block. The complexity of **select** _{S^b} can be exchanged with that of **access** _{S^b} and **partialRank** _{S^b} , by storing explicitly π_b^{-1} rather than π_b .

Note that the individual lower-order terms $o(\sigma)$ needed to support rank and select queries on the bitvectors that encode C^b , and in the structures implemented by Lemma 6, do not necessarily add up to $o(n)$. Thus,

for each of the two cases, we concatenate all the individual bitvectors, we index them for rank and/or select queries, and we simulate operations on each individual bitvector using operations on the bitvectors that result from the concatenation. \square

In some parts of the paper we will need an implementation of **rank** rather than of **partialRank**, and to support this operation efficiently we will use predecessor queries. Given a set of sorted integers, a *predecessor query* returns the index of the largest integer in the set that is smaller than or equal to a given integer provided in input. It is known that predecessor queries can be implemented efficiently, for example with the following data structure:

Lemma 8 ([69, 34]). *Given a sorted sequence of n integers $x^1 < x^2 < \dots < x^n$, where x^i is encoded in $\log U$ bits for all $i \in [1..n]$, we can build in $O(n)$ time and in $O(n \log U)$ bits of working space a data structure that takes $O(n \log U)$ bits of space, and that answers predecessor queries in $O(\log \log U)$ time. This data structure does not require the original sequence of integers to answer queries.*

The original predecessor data structure described in [69] (called *y-fast trie*) has an expected linear time construction algorithm. Construction time is randomized, since the data structure uses a hash table. To obtain deterministic linear construction time, one can replace the hash table with a deterministic dictionary [34].

Implementing rank queries amounts to plugging Lemma 8 into the block partitioning scheme of Lemma 7:

Lemma 9. *Given a string of length n over alphabet $[1..\sigma]$ and an integer $c > 1$, we can build a data structure that takes $n \log \sigma (1 + 1/k) + 6n + O(n/\log^{c-1} \sigma) + o(n)$ bits of space for any positive integer k , and that supports:*

- *either access and partialRank in constant time, select in $O(k)$ time, and rank in $O(kc \log \log \sigma)$ time;*
- *or access and partialRank in $O(k)$ time, select in constant time, and rank in $O(c \log \log \sigma)$ time.*

This data structure can be built in $O(n)$ time and in $o(n)$ bits of working space, and it does not require the original string to support access, rank and select.

Proof. As described in Lemma 7, we divide the string T into blocks of size σ and we build bitvectors \mathbf{freq}_a for every $a \in [1..\sigma]$. We support $\mathbf{rank}_a(i)$ as follows. Let b be the block that contains position i , where blocks are indexed from zero. First, we determine the number of zeros in \mathbf{freq}_a that precede the b -th one, by computing $\mathbf{select}_{\mathbf{freq}_a}(b, 1) - b$. Then, if character a occurs at most $\log^c \sigma$ times inside block b , we binary-search the list of zeros in block b of \mathbf{freq}_a , using at each step a select query to convert the position of a zero inside block b of \mathbf{freq}_a into an occurrence of character a in string T . This process takes $O(\tau c \log \log \sigma)$ time, where τ is the time to perform a select query on T .

If character a occurs more than $\log^c \sigma$ times inside block b of \mathbf{freq}_a , we use a sampling strategy similar to the one described in [69]. Specifically, we sample the relative positions at which a occurs inside block b , every $\log^c \sigma$ occurrences of a zero in \mathbf{freq}_a , and we encode such positions in the data structure described in Lemma 8. Let us call the sampled positions of a block *red positions*, and all other positions *blue positions*. Since positions are relative to a block, the size of the universe is σ , thus the data structure of every block takes $O(m \log \sigma)$ bits of space and it answers queries in time $O(\log \log \sigma)$, where m is the number of red positions of the block. We use the data structure of Lemma 8 to find the index j of the red position of a that immediately precedes position i inside block b : this takes $O(\log \log \sigma)$ time. Since we sampled red positions every $\log^c \sigma$ occurrences of a in block b , we know that there are exactly $(j + 1) \log^c \sigma - 1$ zeros inside block b before the j -th red position. Finally, we find the blue position that immediately precedes position i inside block b by binary-searching the set of $\log^c \sigma - 1$ blue positions between two consecutive red positions, as described above, in time $O(\tau c \log \log \sigma)$.

With this strategy we build $O(n/\log^c \sigma)$ data structures of Lemma 8, containing in total $O(n/\log^c \sigma)$ elements, thus the total space taken by all such data structures is $O(n/\log^{c-1} \sigma)$ bits. Note also that all such

data structures can be built using just $O(\sigma/\log^{c-1} \sigma)$ bits of working space. For every character $a \in [1..\sigma]$, we store all data structures consecutively in memory, and we encode their starting positions in the prefix-sum data structure described in Lemma 4. All such prefix-sum data structures take overall $O(n \log \log \sigma / \log^c \sigma)$ bits of space, and they can be built in $O(\log n)$ bits of working space. We use also a bitvector **which_a** of size $\lceil n/\sigma \rceil$ to mark the blocks of **freq_a** for which we built a data structure of Lemma 8. To locate the starting position of the data structure of a given block and character a , we use a rank query on **which_a** and we query the prefix-sum data structure in constant time. The bitvectors for all characters take overall $n + o(n)$ bits of space. \square

In the space complexity of Lemma 9, we can achieve $5n$ rather than $6n$ by replacing the plain bitvectors **which_a** with the compressed bitvector representation described in [58], which supports constant-time rank queries using $(c \log \log \sigma / \log^c \sigma)n + O(n/\text{polylog}(n))$ bits. Lemma 9 can be further improved by replacing binary searches with queries to the following data structure:

Lemma 10 ([30]). *Given a sorted sequence of n integers $x^1 < x^2 < \dots < x^n$, where x^i is encoded in $\log U$ bits for all $i \in [1..n]$, and given a constant $\epsilon < 1$ and a lookup table of size $O(U^\epsilon)$ bits, we can build in $O(n)$ time and in $O(n \log U)$ bits of working space, a data structure that takes $O(n \log \log U)$ bits of space, and that answers predecessor queries in $O(t/\epsilon)$ time, where t is the time to access an element of the sorted sequence of integers. The lookup table can be built in polynomial time on its size.*

Lemma 11. *Given a string of length n over alphabet $[1..\sigma]$, we can build a data structure that takes $n \log \sigma (1 + 1/k) + O(n \log \log \sigma)$ bits of space for any positive integer k , and that supports:*

- *either access and partialRank in constant time, select in $O(k)$ time, and rank in $O(\log \log \sigma + k)$ time;*
- *or access and partialRank in $O(k)$ time, select in constant time, and rank in $O(\log \log \sigma)$ time.*

This data structure can be built in $O(n)$ time and in $o(n)$ bits of working space, and it does not require the original string to support access, rank and select.

Proof. We proceed as in Lemma 9, but we build the data structure of Lemma 10 on every sequence of consecutive $\log^c \sigma - 1$ blue occurrences of a inside the same block b . Every such data structure uses $O(\log^c \sigma \cdot \log \log \sigma)$ bits of space, and a $O(\tau/\epsilon)$ -time predecessor query to such a data structure replaces the binary search over the blue positions performed in Lemma 10, where τ is the time to perform a select query on T . The total time to build all the data structures of Lemma 10 for all blocks and for all characters is $O(n)$. All such data structures take overall $O(n \log \log \sigma)$ bits of space, and they all share the same lookup table of size $o(\sigma)$ bits, which can be built in $o(\sigma)$ time by choosing ϵ small enough. We also build, in $O(n)$ time, the prefix-sum data structure of Lemma 4, which allows constant-time access to each data structure of Lemma 10. \square

3.5 Representing the topology of suffix trees

It is well known that the topology of an ordered tree T with n nodes can be represented using $2n + o(n)$ bits, as a sequence of $2n$ balanced parentheses built by opening a parenthesis, by recurring on every child of the current node in order, and by closing a parenthesis [49]. To support tree operations on such representation, we will repeatedly use the following data structure:

Lemma 12 ([61, 52]). *Let T be an ordered tree with n nodes, and let $\text{id}(v)$ be the rank of a node v in the preorder traversal of T . Given the balanced parentheses representation of T encoded in $2n + o(n)$ bits, we can build a data structure that takes $2n + o(n)$ bits, and that supports the following operations in constant time:*

- **child(id(v), i):** *returns id(w), where w is the i th child of node v ($i \geq 1$), or \emptyset if v has less than i children;*

- **parent(id(v))**: returns $\text{id}(u)$, where u is the parent of v , or \emptyset if v is the root of T ;
- **lca(id(v), id(w))**: returns $\text{id}(u)$, where u is the lowest common ancestor of nodes v and w ;
- **leftmostLeaf(id(v))**, **rightmostLeaf(id(v))**: returns one plus the number of leaves that, in the preorder traversal of T , are visited before the first (respectively, the last) leaf that belongs to the subtree of T rooted at v ;
- **selectLeaf(i)**: returns $\text{id}(v)$, where v is the i -th leaf visited in the preorder traversal of T ;
- **depth(id(v))**, **height(id(v))**: returns the distance of v from the root or from its deepest descendant, respectively;
- **ancestor(id(v), d)**: returns $\text{id}(u)$, where u is the ancestor of v at depth d ;

This data structure can be built in $O(n)$ time and in $O(n)$ bits of working space.

Note that the operations supported by Lemma 12 are enough to implement a preorder traversal of T in small space, as described by the following folklore lemma:

Lemma 13. *Let T be an ordered tree with n nodes, and let $\text{id}(v)$ be the rank of a node v in the preorder traversal of T . Assume that we have a representation of T that supports the following operations:*

- **firstChild(id(v))**: returns the identifier of the first child of node v in the order of T ;
- **nextSibling(id(v))**: returns the identifier of the child of the parent of node v that follows v in the order of T ;
- **parent(id(v))**: returns the identifier of the parent of node v .

Then, a preorder traversal of T can be implemented using $O(\log n)$ bits of working space.

Proof. During a preorder traversal of T we visit every leaf exactly once, and every internal node exactly twice. Specifically, we visit a node v from its parent, from its previous sibling, or from its last child in the order of T . If we visit v from its parent or from its previous sibling, in the next step of the traversal we will visit the first child of v from its parent – or, if v has no child, we will visit the next sibling of v from its previous sibling if v is not the last child of its parent, otherwise we will visit the parent of v from its last child. If we visit v from its last child, in the next step of the traversal we will visit the next sibling of v from its previous sibling – or, if v has no next sibling, we will visit the parent of v from its last child. Thus, at each step of the traversal we need to store just $\text{id}(v)$ and a single bit that encodes the direction in which we visited v . \square

In this paper we will repeatedly traverse trees in preorder. Not surprisingly, the trees we will be interested in are suffix trees or *contractions* of suffix trees, induced by selecting a subset of the nodes of a suffix tree and by connecting such nodes using their ancestry relationship (the parent of a node in the contracted tree is the nearest selected ancestor in the original tree). We will thus repeatedly need the following space-efficient algorithm for building the balanced parentheses representation of a suffix tree:

Lemma 14. *Let $S \in [1..\sigma]^{n-1}$ be a string. Assume that we are given an algorithm that enumerates all the intervals of $\text{SA}_{S\#}$ that correspond to an internal node of $\text{ST}_{S\#}$, in t time per interval. Then, we can build the balanced parentheses representation of the topology of $\text{ST}_{S\#}$ in $O(nt)$ time and in $O(n)$ bits of working space.*

Proof. We assume without loss of generality that $\log n$ is a power of two. We associate two counters to every position $i \in [1..n]$, one containing the number of open parentheses and the other containing the number of closed parentheses at i . We implement such counters with two arrays $C_o[1..n]$ and $C_c[1..n]$. Given the interval $[i..j]$ of an internal node of $\text{ST}_{S\#}$, we just increment $C_o[i]$ and $C_c[j]$. Once all such intervals have been

enumerated, we scan C_o and C_c synchronously, and for each $i \in [1..n]$ we write $C_o[i] + 1$ open parentheses followed by $C_c[i] + 1$ closed parentheses. The total number of parentheses in the output is at most $2(2n - 1)$.

A naive implementation of this algorithm would use $O(n \log n)$ bits of working space: we achieve $O(n)$ bits using the static allocation strategy described in Section 3.1. Specifically, we partition $C_o[1..n]$ into $\lceil n/b \rceil$ blocks containing $b > 1$ positions each (except possibly for the last block, which might be smaller), and we assign to each block a counter of c bits. Then, we enumerate the intervals of all internal nodes of the suffix tree, incrementing counter $\lceil i/b \rceil$ every time we want to increment position i in C_o . If a counter reaches its maximum value $2^c - 1$, we stop incrementing it and we call *saturated* the corresponding block. The space used by all such counters is $\lceil n/b \rceil \cdot c$ bits, which is $O(n)$ if c is a constant multiple of b . At the end of this process, we allocate a memory area of size $b \log n$ bits to each saturated block, so that every position i in a saturated block has $\log n$ bits available to store $C_o[i]$. Note that there can be at most $(n - 1)/(2^c - 1)$ saturated blocks, so the total memory allocated to saturated blocks is at most $nb \log n / (2^c - 1)$ bits: this quantity is $o(n)$ if 2^c grows faster than $b \log n$.

To every non-saturated block we assign a memory area in which we will store the counters for all the b positions inside the block. Specifically, we will use Elias gamma coding to store a counter value $x \geq 0$ in exactly $1 + 2\lceil \log(x + 1) \rceil \leq 3 + 2\log(x + 1)$ bits [19], and we will concatenate the encodings of all the counters in the same block. The space taken by the memory area of a non-saturated block j whose counter has value $t < 2^c - 1$ is at most:

$$\begin{aligned} & \sum_{i=(j-1)b+1}^{jb} (3 + 2\log(C_o[i] + 1)) \\ & \leq 3b + 2b \log \left(\frac{\sum_{i=(j-1)b+1}^{jb} (C_o[i] + 1)}{b} \right) \end{aligned} \quad (3)$$

$$= 3b + 2b \log \left(\frac{t + b}{b} \right) \quad (4)$$

$$\leq 5b + 2t \quad (5)$$

where Equation 3 derives from applying Jensen's inequality to the logarithm, and Equation 5 comes from the fact that $\log x \leq x$ for all $x \geq 1$. Since $\sum_{i=1}^{\lceil n/b \rceil} t \leq n - 1$, it follows that the total number of bits allocated to non-saturated blocks is at most $7n$ for any choice of b (tighter bounds might be possible, but for clarity we don't consider them here). We concatenate the memory areas of all blocks, and we store a prefix-sum data structure that takes $o(n)$ bits and that returns in constant time the starting position of the memory area allocated to any given block (see Lemma 4). We also store a bitvector `isSaturated`[1.. $\lceil n/b \rceil$] that marks every saturated block with a one and index it for rank queries.

Once memory allocation is complete, we enumerate again the intervals of all internal nodes of the suffix tree, and for every such interval $[i..j]$ we increment $C_o[i]$, as follows. First, we compute the block that contains position i , we use `isSaturated` to determine whether the block is saturated or not, and we use the prefix-sum data structure to retrieve in constant time the starting position of the region of memory assigned to the block. If the block is saturated, we increment the counter that corresponds to position i directly. Otherwise, we access a precomputed table $T_s[1..2^s, 1..b]$ such that $T_s[i, j]$ stores, for every possible configuration i of s bits interpreted as the concatenation of the Elias gamma coding of b counter values x^1, x^2, \dots, x^b , the configuration of s bits that represents the concatenation of the Elias gamma coding of counter values $x^1, x^2, \dots, x^{j-1}, x^j + 1, x^{j+1}, \dots, x^b$. The total number of bits used by all such tables is at most $\sum_{s=1}^y 2^s bs = 2b + b(y - 1)2^{y+1}$, where $y = 3b + 2b \log((t + b)/b)$ with $t = 2^c - 2$ is from Equation 4. Thus, we need to choose b and c so that $by2^y \in o(n)$: setting $b = \log \log n$ and $c = db$ for any constant $d \geq 1$ makes $y \in O((\log \log n)^2)$, and thus $by2^y \in o(n)$. The same choice of b and c guarantees that a cell of T_s can be read in constant time for any s , and that the space for the counters in the memory allocation phase of the algorithm is $O(n)$. Finally, setting $d \geq 2$ guarantees that 2^c grows faster than $b \log n$, thus putting the total memory allocated to *saturated* blocks in $o(n)$. Tables T_s for all $s \in [1..y]$ can be precomputed in time linear in their size.

Array C_c of closed parentheses can be handled in the same way as array C_o . \square

We will also need to be able to follow the suffix link that starts from any node of a suffix tree. Specifically, let operation `suffixLink(id(v))` return the identifier of the destination w of a suffix link from node v of $ST_{S\#}$. The topology of $ST_{S\#}$ can be augmented to support operation `suffixLink`, using just `BWTS#`:

Lemma 15 ([63]). *Let $S \in [1..\sigma]^{n-1}$ be a string. Assume that we are given the representation of the topology of $ST_{S\#}$ described in Lemma 12, the BWT of $S\#$ indexed to support `select` operations in time t , and the C array of S . Then, we can implement function `suffixLink(id(v))` for any node v of $ST_{S\#}$ (possibly a leaf) in $O(t)$ time.*

Proof. Let w be the destination of the suffix link from v , let $[i..j]$ be the interval of node v in $BWT_{S\#}$, and let $\ell(v) = aW$ and $\ell(w) = W$, where $a \in [0..\sigma]$ and $W \in [0..\sigma]^*$. We convert `id(v)` to $[i..j]$ using operations `leftmostLeaf` and `rightmostLeaf` of the topology. Let aWX and aWY be the suffixes of $S\#$ that correspond to positions i and j in $BWT_{S\#}$, respectively, where X and Y are strings on alphabet $[0..\sigma]$. Note that the position i' of WX in $BWT_{S\#}$ is `selecta(BWTS#, i - C[a])`, the position j' of WY in $BWT_{S\#}$ is `selecta(BWTS#, j - C[a])`, and W is the longest prefix of the suffixes that correspond to positions i' and j' in $BWT_{S\#}$, which also labels a node of $ST_{S\#}$. We use operation `selectLeaf` provided by the topology of $ST_{S\#}$ to convert i' and j' to identifiers of leaves in $ST_{S\#}$, and we compute `id(w)` using operation `lca` on such leaves. \square

Note that, if aW is neither a suffix nor a right-maximal substring of $S\#$, i.e. if aW is always followed by the same character $b \in [1..\sigma]$, the algorithm in Lemma 15 maps the locus of aW to the locus of WbX in $ST_{S\#}$, where $X \in [1..\sigma]^*$ and $aWbX$ is the (unique) shortest right-extension of aW that is right-maximal. The locus of WbX might not be the same as the locus of W . As we will see in Section 6, this is the reason why the *bidirectional BWT index* of Definition 7 (on page 36) does not support operation `contractLeft` (respectively, `contractRight`) for strings that are neither suffixes nor right-maximal substrings of $S\#$ (respectively, of S).

3.6 Data structures for monotone minimal perfect hash functions

Given a set $\mathcal{S} \subseteq [1..U]$ of size n , a *monotone minimal perfect hash function* (denoted by MMPHF in what follows) is a function $f : [1..U] \mapsto [1..n]$ such that $x < y$ implies $f(x) < f(y)$ for every $x, y \in \mathcal{S}$. In other words, if the set of elements of \mathcal{S} is $x^1 < x^2 < \dots < x^n$, then $f(x^i) = i$, i.e. the function returns the rank inside \mathcal{S} of the element it takes as an argument. The function is allowed to return an arbitrary value for any $x \in [1..U] \setminus \mathcal{S}$.

To build efficient implementations of MMPHFs, we will repeatedly take advantage of the following lemma:

Lemma 16 ([6]). *Let $\mathcal{S} \subseteq [1..U]$ be a set represented in sorted order by the sequence $x^1 < x^2 < \dots < x^n$, where x^i is encoded in $\log U$ bits for all $i \in [1..n]$ and $\log U < n$. There is an implementation of a MMPHF on \mathcal{S} that takes $O(n \log \log U)$ bits of space, that evaluates $f(x)$ in constant time for any $x \in [1..U]$, and that can be built in randomized $O(n)$ time and in $O(n \log U)$ bits of working space.*

Proof. We use a technique known as *most-significant-bit bucketing* [7]. Specifically, we partition the sequence that represents \mathcal{S} into $\lceil n/b \rceil$ blocks of consecutive elements, where each block $B^i = x^{(i-1)b+1}, \dots, x^{ib}$ contains exactly $b = \log n$ elements (except possibly for the last block $x^{(i-1)b+1}, \dots, x^n$, which might be smaller). Then, we compute the length of the longest common prefix p^i of the elements in every B^i , starting from the most significant bit. To do so, it suffices to compute the longest prefix that is common to the first and to the last element of B^i : this can be done in constant time using the `mostSignificantBit` operation, which can be implemented using a constant number of multiplications [14]. The length of the longest common prefix of a block is at most $\log U - \log \log n \in O(\log U)$.

Then, we build an implementation of a *minimal perfect hash function* F that maps every element in \mathcal{S} onto a number in $[1..n]$. This can be done in $O(n \log U)$ bits of working space and in *randomized* $O(n)$ time: see [33]. We also use a table `lcp[1..n]` that stores at index $F(x^i)$ the length of the longest common prefix

of the block to which x^i belongs, and a table $\text{pos}[1..n]$ that stores at index $F(x^i)$ the relative position of x^i inside its block. Formally:

$$\begin{aligned}\text{lcp}[F(x^i)] &= |p^{\lfloor (i-1)/b \rfloor + 1}| \\ \text{pos}[F(x^i)] &= i - b \cdot \lfloor (i-1)/b \rfloor\end{aligned}$$

The implementation of F takes $O(n + \log \log U)$ bits of space, lcp takes $O(n \log \log U)$ bits, and pos takes $O(n \log \log n)$ bits.

It is folklore that all p^i values are distinct, thus each p^i identifies block i uniquely. We build an implementation of a *minimal perfect* hash function G on set $p^1, p^2, \dots, p^{\lceil n/b \rceil}$, and an inversion table $\text{lcp2block}[1..\lceil n/b \rceil]$ that stores value i at index $G(p^i)$. The implementation of G takes $O(n/\log n + \log \log U)$ bits of space, and it can be built in $O((n/\log n) \log U)$ bits of working space and in randomized $O(n/\log n)$ time. Table lcp2block takes $O((n/\log n) \cdot \log(n/\log n)) = O(n)$ bits. With this setup of data structures, we can return in constant time the rank i in \mathcal{S} of any x^i , by issuing:

$$i = b \cdot \text{lcp2block}\left[G(x^i[1..\text{lcp}[F(x^i)]])\right] + \text{pos}[F(x^i)]$$

where $x^i[g..h]$ denotes the substring of the binary representation of x^i in $\log U$ bits that starts at position g and ends at position h . \square

We will mostly use Lemma 16 inside the following construction, which is based on partitioning the universe rather than the set of numbers:

Lemma 17. *Let $\mathcal{S} \subseteq [1..U]$ be a set represented in sorted order by the sequence $x^1 < x^2 < \dots < x^n$, where x^i is encoded in $\log U$ bits for all $i \in [1..n]$. There is an implementation of a MMPHF on \mathcal{S} that takes $O(n \log \log b) + \lceil U/b \rceil(2 + \lceil \log(nb/U) \rceil) + o(U/b)$ bits of space, that evaluates $f(x)$ in constant time for any $x \in [1..U]$, and that can be built in randomized $O(n)$ time and in $O(b \log b)$ bits of working space, for any choice of b .*

Proof. We will make use of a partitioning technique known as *quotienting* [57]). We partition interval $[1..U]$ into $n' \leq n$ blocks of size b each, except for the last block which might be smaller. Note that the most significant $\log U - \log b$ bits are identical in all elements of \mathcal{S} that belong to the same block. For each block i that contains more than one element of \mathcal{S} , we build an implementation of a monotone minimal perfect hash function f^i on the elements inside the block, as described in Lemma 16, *restricted to their least significant $\log b$ bits*: all such implementations take $O(n \log \log b)$ bits of space in total, and constructing each of them takes $O(b \log b)$ bits of working space. Then, we use Lemma 4 to build a prefix-sum data structure that encodes in $\lceil U/b \rceil(2 + \lceil \log(nb/U) \rceil) + o(U/b)$ bits of space the number of elements in every block. Given an element $x \in [1..U]$, we first find the block it belongs to, by computing $i = \lceil x/b \rceil$, then we use the prefix-sum data structure to compute the number r of elements in \mathcal{S} that belong to blocks smaller than i , and finally we return $r + f^i(x[\log U - \log b + 1..\log U])$, where $x[g..h]$ denotes the substring of the binary representation of x in $\log U$ bits that starts at position g and ends at position h . \square

The construction used in Lemma 17 is a slight generalization of one initially described in [6]. Setting $b = \lceil U/n \rceil$ in Lemma 17 makes the MMPHF implementation fit in $O(n \log \log(U/n))$ bits of space.

3.7 Data structures for range-minimum and range-distinct queries

Given an array of integers $A[1..n]$, let function $\text{rmq}(i, j)$ return an index $k \in [i..j]$ such that $A[k] = \min\{A[x] : x \in [i..j]\}$, with ties broken arbitrarily. We call this function a *range minimum query* (RMQ) over A . It is known that range-minimum queries can be answered by a data structure that is small and efficient to compute:

Lemma 18 ([25]). *Assume that we have a representation of an array of integers $A[1..n]$ that supports accessing the value $A[i]$ stored at any position $i \in [1..n]$ in time t . Then, we can build a data structure that takes $2n + o(n)$ bits of space, and that answers $\text{rmq}(i, j)$ for any pair of integers $i < j$ in $[1..n]$ in constant time, without accessing the representation of A . This data structure can be built in $O(nt)$ time and in $n + o(n)$ bits of working space.*

Assume now that the elements of array $A[1..n]$ belong to alphabet $[1..\sigma]$, and let $\Sigma_{i,j}$ be the set of distinct characters that occur inside subarray $A[i..j]$. Let function $\text{rangeDistinct}(i, j)$ return the set of tuples $\{(c, \text{rank}_A(c, p_c), \text{rank}_A(c, q_c)) : c \in \Sigma_{i,j}\}$ in any order, where p_c and q_c are the first and the last occurrence of c in $A[i..j]$, respectively. The frequency of any $c \in \Sigma_{i,j}$ inside $A[i..j]$ is $\text{rank}_A(c, q_c) - \text{rank}_A(c, p_c) + 1$. It is well known that rangeDistinct queries can be implemented using rmq queries on a specific array, as described in the following lemma:

Lemma 19 ([50, 64, 11]). *Given a string $A \in [1..\sigma]^n$, we can build a data structure of size $n \log \sigma + 8n + o(n)$ bits that answers $\text{rangeDistinct}(i, j)$ for any pair of integers $i < j$ in $[1..n]$ in $O(\text{occ})$ time and in $\sigma \log(n + 1)$ bits of temporary space, where $\text{occ} = |\Sigma_{i,j}|$. This data structure can be built in $O(kn)$ time and in $(n/k) \log \sigma + 2n + o(n)$ bits of working space, for any positive integer k , and it does not require A to answer rangeDistinct queries.*

Proof. To return just the distinct characters in $\Sigma_{i,j}$ it suffices to build a data structure that supports RMQs on an auxiliary array $P[1..n]$, where $P[i]$ stores the position of the *previous occurrence* of character $A[i]$ in A . Since $\text{rmq}(i, j)$ is the leftmost occurrence of character $A[\text{rmq}(i, j)]$ in $A[i..j]$, it is well known that $\Sigma_{i,j}$ can be built by issuing $O(\text{occ})$ rmq queries on P and $O(\text{occ})$ accesses to A , using a stack of $O(\text{occ} \cdot \log n)$ bits and a bitvector of size σ . This is achieved by setting $k = \text{rmq}(i, j)$, by recurring on subintervals $[i..k-1]$ and $[k+1..j]$, and by using the bitvector to mark the distinct characters observed during the recursion and to stop the process if $A[k]$ is already marked [50]. Random access to array P can be simulated in constant time using partialRank and select operations on A , which can be implemented as described in Lemma 7 setting k to a constant. We use the data structures of Lemma 7 also to simulate access to A without storing A itself. We build the RMQ data structure using Lemma 18. After construction, we will never need to answer select queries on A , thus we do not output the $(n/k) \log \sigma + n + o(n)$ bits that encode the inverse permutation in Lemma 7.

To report partial ranks in addition to characters, we adapt this construction as follows. We build a data structure that supports RMQs on an auxiliary array $N[1..n]$, where $N[i]$ stores the position of the *next occurrence* of character $A[i]$ in A . Given an interval $[i..j]$, we first use the RMQ data structure on P and a vector $\text{chars}[1..\sigma]$ of $\sigma \log(n+1)$ bits to store the first occurrence p_c of every $c \in \Sigma_{i,j}$. Then, we use the RMQ data structure on N to detect the last occurrence q_c of every $c \in \Sigma_{i,j}$, and we access $\text{chars}[c]$ both to retrieve the corresponding p_c and to clean up cell $\text{chars}[c]$ for the next query. Finally, we compute $\text{rank}_A(c, p_c)$ and $\text{rank}_A(c, q_c)$ using the partialRank data structure of Lemma 7. To build the data structure that supports RMQs on N , we can use the same memory area of $n + o(n)$ bits used to build the data structure that supports RMQs on P . \square

The temporary space used to answer a rangeDistinct query can be reduced to σ bits by more involved arguments [11]. Rather than using partialRank , select , and access , we can implement the rangeDistinct operation using MMPHFs: the following lemma details this approach, since the rest of the paper will repeatedly use its main technique.

Lemma 20 ([11]). *We can augment a string $A \in [1..\sigma]^n$ with a data structure of size $O(n \log \log \sigma)$ bits that answers $\text{rangeDistinct}(i, j)$ for any pair of integers $i < j$ in $[1..n]$ in $O(\text{occ})$ time and in $\sigma \log(n + 1)$ bits of temporary space, where $\text{occ} = |\Sigma_{i,j}|$. This data structure can be built in $O(n)$ randomized time and in $O(n \log \sigma)$ bits of working space.*

Proof. We build the set of sequences $\{P_c : c \in [1..\sigma]\}$, such that P_c contains all the positions p_1, p_2, \dots, p_k of character c in A in increasing order. We encode P_c as a bitvector such that position p_i for $i > 1$ is represented by the Elias gamma coding of $p_i - p_{i-1}$. The total space taken by all such sequences is $O(n \log \sigma)$ bits, by

applying Jensen's inequality twice. Let $|P_c|$ be the number of bits in P_c : we compute $|P_c|$ and we allocate a corresponding region of memory using the static allocation strategy described in Section 3.1. We also mark in an additional bitvector $\mathbf{start}_c[1..|P_c|]$ the first bit of every representation of a p_i in P_c , and we index \mathbf{start}_c to support **select** queries.

Then, we build an implementation of an MMPHF for every P_c , using Lemma 17 with $U = n$ and $b = \sigma^k$ for some positive integer k . Specifically, for every c , we perform a single scan of sequence P_c , decoding all the positions that fall inside the same block of A of size σ^k , and building an implementation of an MMPHF for the positions inside the block. Once all such MMPHFs have been built, we discard all P_c sequences. The total space used by all MMPHF implementations is at most $O(n(\log \log \sigma + \log k)) + (nk/\sigma^k) \log \sigma + 2n/\sigma^k + o(n/\sigma^k)$ bits: any $k \geq 1$ makes such space fit in $O(n \log \log \sigma)$ bits, and it makes the working space of the construction fit in $O(\sigma^k \log \sigma)$ bits. Since we assumed $\sigma \in o(\sqrt{n}/\log n)$, setting $k \in \{1, 2\}$ makes this additional space fit in $O(n \log \sigma)$ bits.

Finally, we proceed as in Lemma 19. Given a position i , we can compute $\mathbf{rank}_A(A[i], i)$ by querying the MMPHF data structure of character $A[i]$, and we can simulate random access to $P[i]$ by querying the MMPHF data structure of character $A[i]$ and by accessing $p_i - P[i]$ using a **select** operation on $\mathbf{start}_{A[i]}$. \square

Lemma 19 builds an internal representation of A , and the original representation of A provided in the input can be discarded. On the other hand, Lemma 20 uses the input representation of A to answer queries, thus it can be combined with any representation of A that allows constant-time access – for example with those that represent A up to its k th order empirical entropy for $k \in o(\log_\sigma n)$ [24].

4 Enumerating all right-maximal substrings

The following problem lies at the core of our construction and, as we will see in Section 7, it captures the requirements of a number of fundamental string analysis algorithms:

Problem 2. *Given a string $T \in [1..\sigma]^{n-1}\#$, return the following information for all right-maximal substrings W of T :*

- $|W|$ and $\text{range}(W)$ in SA_T ;
- the sorted sequence $b_1 < b_2 < \dots < b_k$ of all the distinct characters in $[0..\sigma]$ such that Wb_i is a substring of T ;
- the sequence of intervals $\text{range}(Wb_1), \dots, \text{range}(Wb_k)$;
- a sequence a_1, a_2, \dots, a_h that lists all the h distinct characters in $[0..\sigma]$ such that a_iW is a prefix of a rotation of T ; the sequence a_1, a_2, \dots, a_h is not necessarily in lexicographic order;
- the sequence of intervals $\text{range}(a_1W), \dots, \text{range}(a_hW)$.

Problem 2 does not specify the order in which the right-maximal substrings of T (or equivalently, the internal nodes of ST_T) must be enumerated, nor the order in which the left-extensions a_iW of a right-maximal substring W must be returned. It does, however, specify the order in which the *right-extensions* Wb_i of W must be returned.

The first step for solving Problem 2 consists in devising a suitable representation for a right-maximal substring W of T . Let $\gamma(a, W)$ be the number of distinct strings Wb such that aWb is a prefix of a rotation of T , where $a \in [0..\sigma]$ and $b \in \{b_1, \dots, b_k\}$. Note that there are precisely $\gamma(a, W)$ distinct characters to the right of aW when it is a prefix of a rotation of T : thus, if $\gamma(a, W) = 0$, then aW is not a prefix of any rotation of T ; if $\gamma(a, W) = 1$ (for example when $a = \#$), then aW is not a right-maximal substring of T ; and if $\gamma(a, W) \geq 2$, then aW is a right-maximal substring of T . This suggests to represent a substring W of T with the following pair:

$$\text{repr}(W) = (\text{chars}[1..k], \text{first}[1..k+1])$$

where $\text{chars}[i] = b_i$ and $\text{range}(Wb_i) = [\text{first}[i].. \text{first}[i+1] - 1]$ for $i \in [1..k]$. Note that $\text{range}(W) = [\text{first}[1].. \text{first}[k+1] - 1]$, since it coincides with the concatenation of the intervals of the right-extensions of W in lexicographic order. If W is not right-maximal, array **chars** and **first** in $\text{repr}(W)$ have length one and two, respectively.

Given $\text{repr}(W)$, $\text{repr}(a_iW)$ can be precomputed for all $i \in [1..h]$, as follows:

Lemma 21. *Assume the notation of Problem 2. Given a data structure that supports **rangeDistinct** queries on BWT_T , given the C array of T , and given $\text{repr}(W) = (\text{chars}[1..k], \text{first}[1..k+1])$ for a substring W of T , we can compute the sequence a_1, \dots, a_h and the corresponding sequence $\text{repr}(a_1W), \dots, \text{repr}(a_hW)$, in $O(t \cdot \text{occ})$ time and in $O(\sigma^2 \log n)$ bits of temporary space, where t is the time taken by the **rangeDistinct** operation per element in its output, and occ is the number of distinct strings a_iWb_j that are the prefix of a rotation of T , where $i \in [1..h]$ and $j \in [1..k]$.*

Proof. Let $\text{leftExtensions}[1..\sigma+1]$ be a vector of characters given in input to the algorithm and initialized to all zeros, and let h be the number of nonempty cells in this vector. We will store in vector **leftExtensions** all characters a_1, a_2, \dots, a_h , not necessarily in lexicographic order. Consider also matrices $A[0..\sigma, 1..\sigma+1]$, $F[0..\sigma, 1..\sigma+1]$ and $L[0..\sigma, 1..\sigma+1]$, given in input to the algorithm and initialized to all zeros, whose rows correspond to possible left-extensions of W . We will store character b_j in cell $A[a_i, p]$, for increasing values of p starting from one, iff a_iWb_j is the prefix of a rotation of T : in this case, we will also set $F[a_i, p] = \text{sp}(a_iWb_j)$ and $L[a_i, p] = \text{ep}(a_iWb_j)$. In other words, every triplet $(A[a_i, p], F[a_i, p], L[a_i, p])$ identifies the right-extension Wb_j of W associated with character $b_j = A[a_i, p]$, and it specifies the interval of a_iWb_j in BWT_T (see Figure 2). We use array **gamma** $[0..\sigma]$, given in input to the algorithm and initialized

to all zeros, to maintain, for every $a \in [0..\sigma]$, the number of distinct characters $b \in \{b_1, \dots, b_k\}$ such that aWb is the prefix of a rotation of T , or equivalently the number of nonempty cells in row a of matrices A , F and L . In other words, $\text{gamma}[a] = \gamma(a, W)$.

For every $j \in [1..k]$, we enumerate all the distinct characters that occur inside the interval $\text{BWT}_T[\text{first}[j]..\text{first}[j+1]-1]$ of string $Wb_j = W \cdot \text{chars}[j]$, along with the corresponding partial ranks, using operation `rangeDistinct`. Recall that `rangeDistinct` does not necessarily return such characters in lexicographic order. For every character a returned by `rangeDistinct`, we compute `range(aWb_j)` in constant time using the C array and the partial ranks, we increment counter `gamma[a]` by one, and we set:

$$\begin{aligned} A[a, \text{gamma}[a]] &= \text{chars}[j] \\ F[a, \text{gamma}[a]] &= \text{sp}(aWb_j) \\ L[a, \text{gamma}[a]] &= \text{ep}(aWb_j) \end{aligned}$$

See Figure 2 for an example. If `gamma[a]` transitioned from zero to one, we increment h by one and we set `leftExtensions[h] = a`. At the end of this process, `leftExtensions[i] = a_i` for $i \in [1..h]$ (note again that the characters in `leftExtensions[1..h]` are not necessarily sorted lexicographically), the nonempty rows in A , F and L correspond to such characters, the characters that appear in row a_i of matrix A are sorted lexicographically, and the corresponding intervals $[F[a_i, p]..L[a_i, p]]$ are precisely the intervals of string $a_i W \cdot A[a_i, p]$ in BWT_T . It follows that such intervals are adjacent in BWT_T , thus:

$$\text{repr}(a_i W) = (A[a_i, 1..\text{gamma}[a_i]], F[a_i, 1..\text{gamma}[a_i]] \bullet (L[a_i, \text{gamma}[a_i]] + 1))$$

where $X \bullet y$ denotes appending number y to the end of array X . We can restore all matrices and vectors to their original state within the claimed time budget, by scanning over all cells of `leftExtensions`, using their value to address matrices A , F and L , and using array `gamma` to determine how many cells must be cleaned in each row of such matrices. \square

Iterated applications of Lemma 21 are almost all we need to solve Problem 2 efficiently, as described in the following lemma:

Lemma 22. *Given a data structure that supports `rangeDistinct` queries on the BWT of a string $T \in [1..\sigma]^{n-1}\#$, and given the C array of T , there is an algorithm that solves Problem 2 in $O(nt)$ time and in $O(\sigma^2 \log^2 n)$ bits of working space, where t is the time taken by the `rangeDistinct` operation per element in its output.*

Proof. We use again the notation of Problem 2. Assume by induction that we know $\text{repr}(W) = (\text{chars}[1..k], \text{first}[1..k+1])$ and $|W|$ for some right-maximal substring W of T . Using Lemma 21, we compute a_i and $\text{repr}(a_i W) = (\text{chars}_i[1..k_i], \text{first}_i[1..k_i+1])$ for all $i \in [1..h]$, and we determine whether $a_i W$ is right-maximal by checking whether $|\text{chars}_i| > 1$, or equivalently whether `gamma[a_i] > 1` in Lemma 21: if this is the case, we push pair $(\text{repr}(a_i W), |W| + 1)$ to a stack S . In the next iteration, we pop the representation of a string from the stack and we repeat the process, until the stack becomes empty. Note that this is equivalent to following all the explicit Weiner links from (or equivalently, all the reverse suffix links to) the node v of ST_T with $\ell(v) = W$, not necessarily in lexicographic order. Thus, running the algorithm from a stack initialized with $\text{repr}(\varepsilon)$ is equivalent to a depth-first traversal of the *suffix-link tree* of T (not necessarily following the lexicographic order of Weiner link labels): recall from Section 2.3 that a traversal of SLT_T guarantees to enumerate all the right-maximal substrings of T . Triplet $\text{repr}(\varepsilon)$ can be easily built from the C array of T .

Every `rangeDistinct` query performed by the algorithm can be charged to a distinct node of ST_T , and every tuple in the output of all such `rangeDistinct` queries can be charged to a distinct (explicit or implicit) Weiner link. It follows from Observation 1 that the algorithm runs in $O(nt)$ time. Since the algorithm performs a depth-first traversal of the suffix-link tree of T , the depth of the stack is bounded by the length of a longest right-maximal substring of T . More precisely, since we always pop the element at the top of the stack, the depth of the stack is bounded by quantity μ_T defined in Section 2.2, i.e. by the largest number of (not necessarily proper) suffixes of a maximal repeat that are themselves maximal repeats. Even

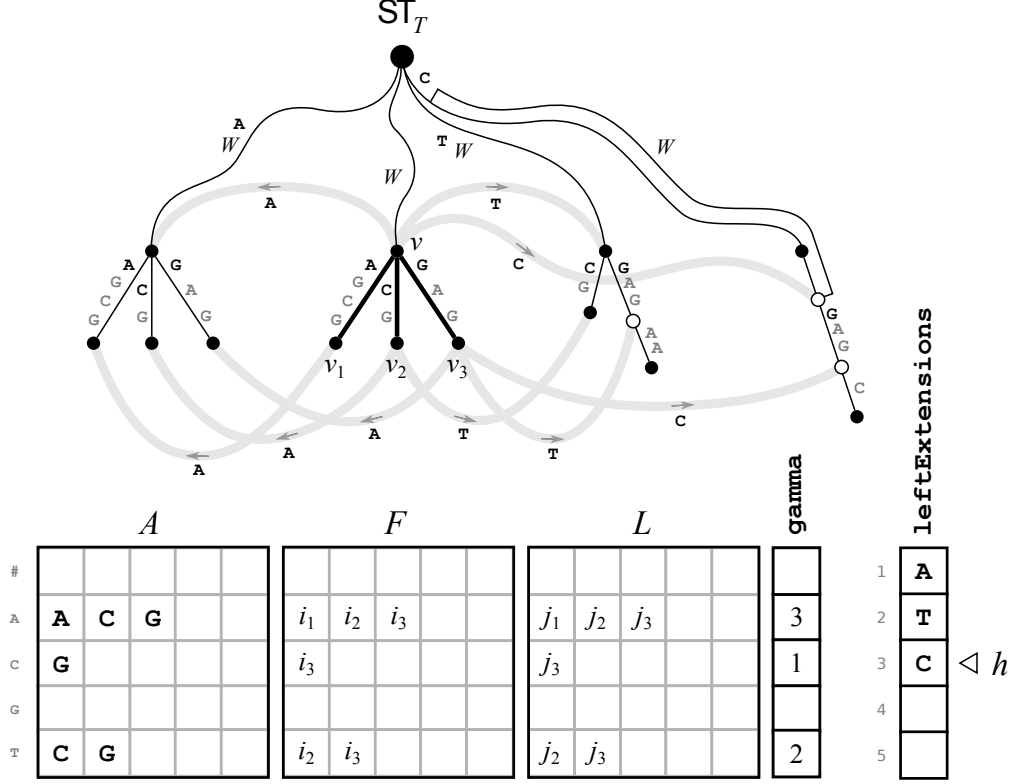


Figure 2: Lemma 21 applied to the right-maximal substring $W = \ell(v)$. Gray directed arcs represent implicit and explicit Weiner links. White dots represent the destinations of implicit Weiner links. Child v_k of node v in ST_T has interval $[i_k..j_k]$ in BWT_T , where $k \in [1..3]$. Among all strings prefixed by string W , only those prefixed by $WGAG$ are preceded by C : it follows that CW is always followed by G and it is not right-maximal, thus the Weiner link from v labeled by C is implicit. Conversely, $WAGCG$, WCG and $WGAG$ are all preceded by an A , so AW is right-maximal and the Weiner link from v labeled by A is explicit.

more precisely, since we push just right-maximal substrings, the depth of the stack is bounded by quantity λ_T defined in Section 2.2. Unfortunately, λ_T might be $O(n)$. We reduce this depth to $O(\log n)$ by pushing at every iteration the pair $(\text{repr}(a_iW), |a_iW|)$ with largest $\text{range}(a_iW)$ first (a technique already described in [36]): the interval of every other aW is necessarily at most half of $\text{range}(W)$, thus stack S contains at any time pairs from $O(\log n)$ suffix-link tree levels. Every such level contains $O(\sigma)$ pairs, and every pair takes $O(\sigma \log n)$ bits, thus the total space used by the stack is $O(\sigma^2 \log^2 n)$ bits. \square

Algorithm 2 summarizes Lemma 22 in pseudocode. Combining Lemma 22 with the **rangeDistinct** data structure of Lemma 19 we obtain the following result:

Theorem 3. *Given the BWT of a string $T \in [1..\sigma]^{n-1}\#$, we can solve Problem 2 in $O(nk)$ time, and in $n \log \sigma(1 + 1/k) + 10n + (2\sigma + 1) \log(n + 1) + o(n) = n \log \sigma(1 + 1/k) + O(n) + O(\sigma \log n)$ bits of working space, for any positive integer k .*

Proof. Lemma 22 needs just the C array, which takes $(\sigma + 1) \log n$ bits, and a **rangeDistinct** data structure: the one in Lemma 19 takes $n \log \sigma + 8n + o(n)$ bits of space, and it answers queries in time linear in the size of their output and in $\sigma \log(n + 1)$ bits of space in addition to the output. Building the C array from BWT_T takes $O(n)$ time, and building the **rangeDistinct** data structure of Lemma 19 takes $O(nk)$ time and $(n/k) \log \sigma + 2n + o(n)$ bits of working space, for any positive integer k . \square

Note that replacing Lemma 19 in Theorem 3 with the alternative construction of Lemma 20 introduces randomization and it does not improve space complexity.

As we saw in Section 3.5, having an efficient algorithm to enumerate all intervals in BWT_T of right-maximal substrings of T has an immediate effect on the construction of the balanced parentheses representation of ST_T . The following result derives immediately from plugging Theorem 3 in Lemma 14:

Theorem 4. *Given the BWT of a string $T \in [1..\sigma]^{n-1}\#$, we can build the balanced parentheses representation of the topology of ST_T in $O(nk)$ time and in $n \log \sigma(1 + 1/k) + O(n)$ bits of working space, for any positive integer k .*

In this paper we will also need to enumerate all the right-maximal substrings of a concatenation $T = T^1 T^2 \dots T^m$ of m strings T^1, T^2, \dots, T^m , where $T^i \in [1..\sigma]^{n_i-1}\#_i$ for $i \in [1..m]$. Recall that the right-maximal substrings of T correspond to the internal nodes of the *generalized suffix tree* of T^1, T^2, \dots, T^m , thus we can solve Problem 5 by applying Lemma 22 to the BWT of T . If we are just given the BWT of each T^i separately, however, we can represent a substring W as a pair of sets of arrays $\text{repr}'(W) = (\{\text{chars}^1, \dots, \text{chars}^m\}, \{\text{first}^1 \dots \text{first}^m\})$, where chars^i collects all the distinct characters b such that Wb is observed in string T^i , in lexicographic order, and the interval of string $W \cdot \text{chars}^i[j]$ in BWT_{T^i} is $[\text{first}^i[j].. \text{first}^i[j+1] - 1]$. If W does not occur in T^i , we assume that $|\text{chars}^i| = 0$ and that $\text{first}^i[1]$ equals one plus the number of suffixes of T^i that are lexicographically smaller than W . If necessary, this representation can be converted in $O(m\sigma)$ time into a representation based on intervals of BWT_T . We can thus adapt the approach of Lemma 22 to solve the following generalization of Problem 2, as described in Lemma 23 below:

Problem 5. *Given strings T^1, T^2, \dots, T^m with $T^i \in [1..\sigma]^{n_i-1}\#_i$ for $i \in [1..m]$, return the following information for all right-maximal substrings W of $T = T^1 T^2 \dots T^m$:*

- $|W|$ and $\text{range}(W)$ in SA_T ;
- the sorted sequence $b_1 < b_2 < \dots < b_k$ of all the distinct characters in $[-m+1..\sigma]$ such that Wb_i is a substring of T ;
- the sequence of intervals $\text{range}(Wb_1), \dots, \text{range}(Wb_k)$;
- a sequence a_1, a_2, \dots, a_h that lists all the h distinct characters in $[-m+1..\sigma]$ such that $a_i W$ is the prefix of a rotation of T ; the sequence a_1, a_2, \dots, a_h is not necessarily in lexicographic order;
- the sequence of intervals $\text{range}(a_1 W), \dots, \text{range}(a_h W)$.

Lemma 23. *Assume that we are given a data structure that supports `rangeDistinct` queries on the BWT of a string T^i , and the C array of T^i , for all strings in a set $\{T^1, T^2, \dots, T^m\}$, where $T^i \in [1..\sigma]^{n_i-1}\#_i$ for $i \in [1..m]$. There is an algorithm that solves Problem 5 in $O(mnt)$ time and in $O(m\sigma^2 \log^2 n)$ bits of working space, where t is the time taken by the `rangeDistinct` operation per element in its output, and $n = \sum_{i=1}^m n_i$.*

Proof. To keep the presentation as simple as possible we omit the details on how to handle strings that occur in some T^i but that do not occur in some T^j with $j \neq i$. We use the same algorithm as in Lemma 22, but this time with the following data structures:

- m distinct arrays $\text{gamma}^1, \text{gamma}^2, \dots, \text{gamma}^m$;
- m distinct matrices $A^1, A^2, \dots, A^m, F^1, F^2, \dots, F^m$, and L^1, L^2, \dots, L^m ;
- a single stack, in which we push $\text{repr}'(W)$ tuples;
- a single array $\text{leftExtensions}[1..\sigma + m]$, which stores all the distinct left-extensions of a string W that are the prefix of a rotation of a string T^i , not necessarily in lexicographic order.

Given $\text{repr}'(W)$ for a right-maximal substring W of T , we apply Lemma 21 to each T^i to compute the corresponding $\text{repr}(aW)$ for all strings aW that are the prefix of a rotation of T^i , updating row a in A^i , F^i , L^i and gamma^i accordingly, and adding a character a to the shared array `leftExtensions` whenever we see a for the first time in any T^i (see Algorithm 4). If $a = \#_i$, we assume it is actually $\#_{i-1}$ if $i > 1$, and we assume it is $\#_m$ if $i = 1$. We push to the shared stack the pair $\text{repr}'(aW) = (\{\text{chars}^1 \dots \text{chars}^m\}, \{\text{first}^1 \dots \text{first}^m\})$ such that $\text{chars}^i = A^i[a, 1.. \text{gamma}^i[a]]$, $\text{first}^i = F^i[a, 1.. \text{gamma}^i[a]] \bullet (L^i[a, \text{gamma}^i[a]] + 1)$ for all $i \in [1..m]$, if and only if aW is right-maximal in T , or equivalently iff there is an $i \in [1..m]$ such that $\text{gamma}^i[a] > 1$, or alternatively if there are two integers $i \neq j$ in $[1..m]$ such that $\text{gamma}^i[a] = 1$, $\text{gamma}^j[a] = 1$, and $A^i[a][1] \neq A^j[a][1]$ (see Algorithm 3). Note that we never push $\text{repr}'(aW)$ with $a = \#_i$ in the stack, thus the space taken by the stack is $O(m\sigma^2 \log^2 n)$ bits. In analogy to Lemma 22, we push first to the stack the left-extension aW of W that maximizes $\sum_{i=1}^m |\mathbb{I}(aW, T^i)| = \sum_{i=1}^m L^i[a, \text{gamma}^i[a]] - F^i[a, 1] + 1$. The result of this process is a traversal of the suffix-link tree of T , not necessarily following the lexicographic order of its Weiner link labels. The total cost of translating every $\text{repr}'(W)$ into the quantities required by Problem 5 is $O(mn)$. \square

Recall that we say that a node (possibly a leaf) of the suffix tree of $T = T^1 T^2 \dots T^m$ is *pure* if all the leaves in its subtree are suffixes of exactly one string T^i , and we call it *impure* otherwise. Lemma 23 can be adapted to traverse only impure nodes of the generalized suffix tree. This leads to the following algorithm for building the BWT of T from the BWT of T^1, T^2, \dots, T^m :

Lemma 24. *Assume that we are given a data structure that supports `rangeDistinct` queries on the BWT of a string T^i , and the C array of T^i , for all strings in a set $\{T^1, T^2, \dots, T^m\}$, where $T^i \in [1..\sigma]^{n_i-1} \#_i$ for $i \in [1..m]$. There is an algorithm that builds the BWT of string $T = T^1 T^2 \dots T^m$ in $O(mnt)$ time and in $O(m\sigma^2 \log^2 n)$ bits of working space, where t is the time taken by the `rangeDistinct` operation per element in its output, and $n = \sum_{i=1}^m n_i$.*

Proof. The BWT of T can be partitioned into disjoint intervals that correspond to *pure nodes of minimal depth* in ST_T , i.e. to pure nodes whose parent is impure. In ST_T , suffix links from impure nodes lead to other impure nodes, so the set of all impure nodes is a subgraph of the suffix-link tree of T , it includes the root, and it can be traversed by iteratively taking explicit Weiner links from the root. We modify Algorithm 3 to traverse only impure internal nodes of ST_T , by pushing to the stack $\text{repr}'(aW) = (\{\text{chars}^i\}, \{\text{first}^i\})$, where $\text{chars}^i = A^i[a, 1.. \text{gamma}^i[a]]$ and $\text{first}^i = F^i[a, 1.. \text{gamma}^i[a]] \bullet (L^i[a, \text{gamma}^i[a]] + 1)$ for all $i \in [1..m]$, iff it represents an internal node of ST_T , and moreover if there are two integers $i \neq j$ in $[1..m]$ such that $\text{gamma}^i[a] > 0$ and $\text{gamma}^j[a] > 0$.

Assume that we enumerate an impure internal node of ST_T with label W , and let $\text{repr}'(W) = (\{\text{chars}^i\}, \{\text{first}^i\})$. We merge in linear time the set of sorted arrays $\{\text{chars}^i\}$. Assume that character $b = \text{chars}^i[j]$ occurs only in chars^i . It follows that the locus of Wb in ST_T is a pure node of minimal depth, and we can copy $\text{BWT}_{T^i}[\text{first}^i[j].. \text{first}^i[j+1] - 1]$ to $\text{BWT}_T[x..x + \text{first}^i[j+1] - \text{first}^i[j] - 1]$, where $x = 1 + \sum_{i=1}^m \text{smaller}(b, i)$ and

$$\text{smaller}(b, i) = \begin{cases} \text{first}^i[1] - 1 & \text{if } (|\text{chars}^i| = 0) \text{ or } (\text{chars}^i[1] \geq b) \\ \max_{j: \text{chars}^i[j] < b} \{\text{first}^i[j+1] - 1\} & \text{otherwise} \end{cases}$$

The value of x can be easily maintained while merging set $\{\text{chars}^i\}$. If character b occurs in more than one chars^i array, then the locus of Wb in ST_T is impure, and it will be enumerated (or it has already been enumerated) by the traversal algorithm. \square

In the rest of the paper we will focus on the case $m = 2$. The following theorem, which we will use extensively in Section 7, combines Lemma 23 for $m = 2$ with the `rangeDistinct` data structure of Lemma 19:

Theorem 6. *Given the BWT of a string $S^1 \in [1..\sigma]^{n_1-1} \#_1$ and the BWT of a string $S^2 \in [1..\sigma]^{n_2-1} \#_2$, we can solve Problem 5 in $O(nk)$ time and in $n \log \sigma(1 + 1/k) + 10n + o(n)$ bits of working space, for any positive integer k , where $n = n_1 + n_2$.*

Finally, in Section 5 we will work on strings that are not terminated by a special character, thus we will need the following version of Lemma 24 that works on sets of rotations rather than on sets of suffixes:

Theorem 7. *Let $S^1 \in [1..\sigma]^{n_1}$ and $S^2 \in [1..\sigma]^{n_2}$ be two strings such that $|\mathcal{R}(S^1)| = n_1$, $|\mathcal{R}(S^2)| = n_2$, and $\mathcal{R}(S^1) \cap \mathcal{R}(S^2) = \emptyset$. Given the BWT of $\mathcal{R}(S^1)$ and the BWT of $\mathcal{R}(S^2)$, we can build the BWT of $\mathcal{R}(S^1) \cup \mathcal{R}(S^2)$ in $O(nk)$ time and in $n \log \sigma(1 + 1/k) + 10n + o(n)$ bits of working space, where $n = n_1 + n_2$.*

Proof. Since all rotations of S^i are lexicographically distinct, the compact trie of all such rotations is well defined, and every leaf of such trie corresponds to a distinct rotation of S^i . Since no rotation of S^1 is lexicographically identical to a rotation of S^2 , the generalized compact trie that contains all rotations of S^1 and all rotations of S^2 is well defined, and every leaf of such trie corresponds to a distinct rotation of S^1 or of S^2 . We can thus traverse such generalized compact trie using BWT_{S^1} and BWT_{S^2} as described in Lemma 24, using Lemma 19 to implement **rangeDistinct** data structures. \square

ALGORITHM 1: Building $\text{repr}(aW)$ from $\text{repr}(W)$ for all $a \in [0..\sigma]$ such that aW is a prefix of a rotation of $T \in [1..\sigma]^{n-1}\#$.

Input: $\text{repr}(W)$ for a substring W of T . Support for **rangeDistinct** queries on the BWT of T . C array of T .

Empty matrices A , F , and L , empty arrays **gamma** and **leftExtensions**, and a pointer h .

Output: Matrices A , F , L , pointer h , arrays **gamma** and **leftExtensions**, filled as described in Lemma 22.

```

1 (chars, first)  $\leftarrow$  repr( $W$ );
2  $h \leftarrow 0$ ;
3 for  $j \in [1..|\text{chars}|]$  do
4    $\mathcal{I} \leftarrow \text{BWT}_T.\text{rangeDistinct}(\text{first}[j], \text{first}[j+1] - 1)$ ;
5   for  $(a, p_a, q_a) \in \mathcal{I}$  do
6     if gamma[ $a$ ] = 0 then
7        $h \leftarrow h + 1$ ;
8       leftExtensions[ $h$ ]  $\leftarrow a$ ;
9     end
10    gamma[ $a$ ]  $\leftarrow$  gamma[ $a$ ] + 1;
11     $A[a, \text{gamma}[a]] \leftarrow \text{chars}[j]$ ;
12     $F[a, \text{gamma}[a]] \leftarrow C[a] + p_a$ ;
13     $L[a, \text{gamma}[a]] \leftarrow C[a] + q_a$ ;
14  end
15 end
```

ALGORITHM 2: Enumerating all right-maximal substrings of $T \in [1..\sigma]^{n-1}\#$. See Lemma 21 for a definition of operator \bullet . The callback function `callback` highlighted in gray just prints the pair $(\text{repr}(W), |W|)$ given in input. Section 7 describes other implementations of `callback`.

Input: BWT transform and C array of T . Array `distinctChars` of all the distinct characters that occur in T , in lexicographic order, and array `start` of starting positions of the corresponding intervals in BWT_T . Support for `rangeDistinct` queries on BWT_T , and an implementation of Algorithm 1 (function `extendLeft`).

Output: $(\text{repr}(W), |W|)$ for all right-maximal substrings W of T .

```

1   $S \leftarrow$  empty stack;
2   $A \leftarrow \text{zeros}[0..\sigma, 1..\sigma + 1]$ ;
3   $F \leftarrow \text{zeros}[0..\sigma, 1..\sigma + 1]$ ;
4   $L \leftarrow \text{zeros}[0..\sigma, 1..\sigma + 1]$ ;
5   $\text{gamma} \leftarrow \text{zeros}[0..\sigma]$ ;
6   $\text{leftExtensions} \leftarrow \text{zeros}[1..\sigma + 1]$ ;
7   $\text{repr}(\varepsilon) \leftarrow (\text{distinctChars}, \text{start} \bullet (n + 1))$ ;
8   $S.\text{push}((\text{repr}(\varepsilon), 0))$ ;
9  while not  $S.\text{isEmpty}()$  do
10      $(\text{repr}(W), |W|) \leftarrow S.\text{pop}()$ ;
11      $h \leftarrow 0$ ;
12      $\text{extendLeft}(\text{repr}(W), \text{BWT}_T, C, A, F, L, \text{gamma}, \text{leftExtensions}, h)$ ;
13     callback $(\text{repr}(W), |W|, \text{BWT}_T, C, A, F, L, \text{gamma}, \text{leftExtensions}, h)$ ;
    /* Pushing right-maximal left-extensions on the stack */
14      $C \leftarrow \{c : c = \text{leftExtensions}[i], i \in [1..h], \text{gamma}[c] > 1\}$ ;
15     if  $C \neq \emptyset$  then
16          $c \leftarrow \text{argmax}\{L[c, \text{gamma}[c]] - F[c, 1] : c \in C\}$ ;
17          $\text{repr}(cW) \leftarrow (A[c, 1..\text{gamma}[c]], F[c, 1..\text{gamma}[c]] \bullet (L[c, \text{gamma}[c]] + 1))$ ;
18          $S.\text{push}(\text{repr}(cW), |W| + 1)$ ;
19         for  $a \in C \setminus \{c\}$  do
20              $\text{repr}(aW) \leftarrow (A[a, 1..\text{gamma}[a]], F[a, 1..\text{gamma}[a]] \bullet (L[a, \text{gamma}[a]] + 1))$ ;
21              $S.\text{push}(\text{repr}(aW), |W| + 1)$ ;
22         end
23     end
    /* Cleaning up for the next iteration */
24     for  $i \in [1..h]$  do
25          $a \leftarrow \text{leftExtensions}[i]$ ;
26         for  $j \in [1..\text{gamma}[a]]$  do
27              $A[a, j] \leftarrow 0$ ;
28              $F[a, j] \leftarrow 0$ ;
29              $L[a, j] \leftarrow 0$ ;
30         end
31          $\text{gamma}[a] \leftarrow 0$ ;
32     end
33 end

```

ALGORITHM 3: Enumerating all right-maximal substrings of $T = T^1\#_1T^2\#_2\cdots T^m\#_m$, where $T^i \in [1..\sigma]^{n_i-1}$ for $i \in [1..m]$, $m \geq 1$. The key differences from Algorithm 2 are highlighted in gray. To iterate over all *impure* right-maximal substrings of T , it suffices to replace just the gray lines (see Lemma 24). See Lemma 21 for a definition of operator \bullet . The callback function `callback` just prints its input ($\text{repr}'(W), |W|$). For brevity the case in which a string occurs in some T^i but does not occur in some T^j is not handled.

Input: BWT transform and C array of string $T^i\#$. Array distinctChars^i of all the distinct characters that occur in $T^i\#$, in lexicographic order, and array start^i of starting positions of the corresponding intervals in $\text{BWT}_{T^i\#}$. Support for `rangeDistinct` queries on $\text{BWT}_{T^i\#}$, and an implementation of Algorithm 4 (function `extendLeft'`).

Output: ($\text{repr}'(W), |W|$) for all right-maximal substrings W of T .

```

1   $S \leftarrow$  empty stack;
2  for  $i \in [1..m]$  do
3     $A^i \leftarrow \text{zeros}[0..\sigma, 1..\sigma + 1]$ ,  $F^i \leftarrow \text{zeros}[0..\sigma, 1..\sigma + 1]$ ;
4     $L^i \leftarrow \text{zeros}[0..\sigma, 1..\sigma + 1]$ ,  $\text{gamma}^i \leftarrow \text{zeros}[0..\sigma]$ ;
5  end
6   $\text{leftExtensions} \leftarrow \text{zeros}[1..\sigma + m]$ ;
7   $\text{seen} \leftarrow \text{zeros}[1..\sigma]$ ;
8   $\text{repr}'(\varepsilon) \leftarrow (\{\text{distinctChars}^i\}, \{\text{start}^i \bullet (n_i + 1)\})$ ;
9   $S.\text{push}((\text{repr}'(\varepsilon), 0))$ ;
10 while not  $S.\text{isEmpty}()$  do
11    $(\text{repr}'(W), |W|) \leftarrow S.\text{pop}()$ ;
12    $h \leftarrow 0$ ;
13    $\text{extendLeft}'(\text{repr}'(W), \{\text{BWT}_{T^i}\}, \{C^i\}, \{A^i\}, \{F^i\}, \{L^i\}, \{\text{gamma}^i\}, \text{leftExtensions}, \text{seen}, h)$ ;
14    $\text{callback}(\text{repr}'(W), |W|, \{\text{BWT}_{T^i}\}, \{C^i\}, \{A^i\}, \{F^i\}, \{L^i\}, \{\text{gamma}^i\}, \text{leftExtensions}, h)$ ;
15   /* Pushing right-maximal left-extensions on the stack */
16    $\mathcal{C} \leftarrow \{c > 0 : c = \text{leftExtensions}[i], i \in [1..h], (\exists p \in [1..m] : \text{gamma}^p[c] > 1)$ 
17    $\text{or } (\exists p \neq q : \text{gamma}^p[c] = 1, \text{gamma}^q[c] = 1, A^p[c, 1] \neq A^q[c, 1])\}$ ;
18   if  $\mathcal{C} \neq \emptyset$  then
19      $c \leftarrow \text{argmax} \{\sum_{i=1}^m L^i[c, \text{gamma}^i[c]] - F^i[c, 1] : c \in \mathcal{C}\}$ ;
20      $\text{repr}'(cW) \leftarrow (\{A^i[c, 1..\text{gamma}^i[c]]\}, \{F^i[c, 1..\text{gamma}^i[c]] \bullet (L^i[c, \text{gamma}^i[c]] + 1)\})$ ;
21      $S.\text{push}(\text{repr}'(cW), |W| + 1)$ ;
22     for  $a \in \mathcal{C} \setminus \{c\}$  do
23        $\text{repr}'(aW) \leftarrow (\{A^i[a, 1..\text{gamma}^i[a]]\}, \{F^i[a, 1..\text{gamma}^i[a]] \bullet (L^i[a, \text{gamma}^i[a]] + 1)\})$ ;
24        $S.\text{push}(\text{repr}'(aW), |W| + 1)$ ;
25     end
26   end
27   /* Cleaning up for the next iteration */
28   for  $i \in [1..h]$  do
29      $a \leftarrow \text{leftExtensions}[i]$ ;
30     if  $a \leq 0$  then
31        $k \leftarrow -a + 2 \pmod{m}$ ;
32        $A^k[0, 1] \leftarrow 0$ ,  $F^k[0, 1] \leftarrow 0$ ,  $L^k[0, 1] \leftarrow 0$ ,  $\text{gamma}^k[0] \leftarrow 0$ ;
33     end
34     else
35        $\text{seen}[a] \leftarrow 0$ ;
36       for  $j \in [1..m]$  do
37         for  $k \in [1..\text{gamma}^j[a]]$  do
38            $A^j[a, k] \leftarrow 0$ ,  $F^j[a, k] \leftarrow 0$ ,  $L^j[a, k] \leftarrow 0$ ;
39         end
40          $\text{gamma}^j[a] \leftarrow 0$ ;
41       end
42     end
43   end
44 end

```

ALGORITHM 4: Building $\text{repr}'(aW)$ from $\text{repr}'(W)$ for all $a \in [-m + 1.. \sigma]$ such that aW is a prefix of a rotation of $T = T^1 \#_1 T^2 \#_2 \dots T^m \#_m$, where $m \geq 1$ and $T^i \in [1..\sigma]^{n_i-1}$. The lines highlighted in gray are the key differences from Algorithm 1.

Input: $\text{repr}'(W)$ for a substring W of T . Support for `rangeDistinct` queries on $\text{BWT}_{T^i \#}$, C array of T^i , empty matrices A^i , F^i and L^i , and empty array gamma^i of string T^i , for all $i \in [1..m]$. A single empty array `leftExtensions`, a single bitvector `seen`, and a single pointer h .

Output: Matrices A^i , F^i , L^i , pointer h , and arrays gamma^i and `leftExtensions`, for all $i \in [1..m]$, filled as described in Lemma 23.

```

1  ( $\{\text{chars}^i\}, \{\text{first}^i\}$ )  $\leftarrow \text{repr}'(W)$ ;
2   $h \leftarrow 0$ ;
3  for  $i \in [1..m]$  do
4      for  $j \in [1..\text{chars}^i]$  do
5           $\mathcal{I} \leftarrow \text{BWT}_{T^i \#}.\text{rangeDistinct}(\text{first}^i[j], \text{first}^i[j+1] - 1)$ ;
6          for  $(a, p_a, q_a) \in \mathcal{I}$  do
7              if  $a = 0$  then
8                   $h \leftarrow h + 1$ ;
9                   $\text{leftExtensions}[h] \leftarrow -(i - 1 \pmod{m}) + 1$ ;
10             end
11             else
12                 if seen[ $a$ ] = 0 then
13                     seen[ $a$ ] = 1;
14                      $h \leftarrow h + 1$ ;
15                      $\text{leftExtensions}[h] \leftarrow a$ ;
16                 end
17             end
18              $\text{gamma}^i[a] \leftarrow \text{gamma}^i[a] + 1$ ;
19              $A^i[a, \text{gamma}^i[a]] \leftarrow \text{chars}^i[j]$ ;
20              $F^i[a, \text{gamma}^i[a]] \leftarrow C^i[a] + p_a$ ;
21              $L^i[a, \text{gamma}^i[a]] \leftarrow C^i[a] + q_a$ ;
22         end
23     end
24 end

```

5 Building the Burrows-Wheeler transform

It is well-known that the Burrows-Wheeler transform of a string $T\#$ such that $T \in [1..\sigma]^n$ and $\# = 0 \notin [1..\sigma]$, can be built in $O(n \log \log \sigma)$ time and in $O(n \log \sigma)$ bits of working space [38]. In this section we bring construction time down to $O(n)$ by plugging Theorem 7 into the recursive algorithm described in [38], which we summarize here for completeness.

Specifically, we partition T into blocks of equals size B . For convenience, we work with a version of T whose length is a multiple of B , by appending to the end of T the smallest number of occurrences of character $\#$ such that the length of the resulting padded string is an integer multiple of B , and such that the padded string contains at least one occurrence of $\#$. Recall that $B \cdot \lceil x/B \rceil$ is the smallest multiple of B that is at least x . Thus, we append $n' - n$ copies of character $\#$ to T , where $n' = B \cdot \lceil (n+1)/B \rceil$. To simplify notation we call the resulting string X , and we use n' to denote the length of X .

We interpret a partitioning of X into blocks as a new string X_B of length n'/B , defined on the alphabet $[1..(\sigma+1)^B]$ of all strings of length B on alphabet $[0..\sigma]$: the “characters” of X_B correspond to the blocks of X . In other words, $X_B[i] = X[(i-1)B+1..iB]$. We assume B to be even, and we denote by **left** (respectively, **right**) the function from $[1..(\sigma+1)^B]$ to $[1..(\sigma+1)^{B/2}]$ such that **left**(W) returns the first (respectively, the second) half of block W . In other words, if $W = w_1 \cdots w_B$, **left**(W) = $w_1 \cdots w_{B/2}$ and **right**(W) = $w_{B/2+1} \cdots w_B$. We also work with circular rotations of X (see Section 2.2): specifically, we denote by \overleftarrow{X} string $X[B/2+1..n'] \cdot X[1..B/2]$, or equivalently string X *circularly rotated to the left by $B/2$ positions*, and we denote by \overleftarrow{X}_B the string on alphabet $[1..(\sigma+1)^B]$ induced by partitioning \overleftarrow{X} into blocks of size B .

Note that the suffix that starts at position i in \overleftarrow{X}_B equals the half-block $P_i = X[B/2+(i-1)B+1..iB]$, followed by string $S_i = F_{i+1} \cdot X[1..B/2]$, where F_{i+1} is the suffix of X_B that starts at position $i+1$ in X_B , if any. Thus, it is not surprising that we can derive the BWT of string \overleftarrow{X}_B from the BWT of string X_B :

Lemma 25 ([38]). *The BWT of string \overleftarrow{X}_B can be derived from the BWT of string X_B in $O(n'/B)$ time and $O(\sigma^B \cdot \log(n'/B))$ bits of working space, where $n' = |X|$.*

The second key observation that we exploit for building the BWT of X is the fact that the suffixes of $X_{B/2}$ which start at odd positions coincide with the suffixes of X_B , and the suffixes of $X_{B/2}$ that start at even positions coincide with the suffixes of \overleftarrow{X}_B . Thus, we can reconstruct the BWT of $X_{B/2}$ by merging the BWT of X_B with the BWT of \overleftarrow{X}_B : this is where Theorem 7 comes into play.

Lemma 26. *Assume that we can read in constant time a block of B characters. Then, the BWT of string $X_{B/2}$ can be derived from the BWT of string X_B and from the BWT of string \overleftarrow{X}_B , in $O(n'/B)$ time and $O(n' \log \sigma)$ bits of working space, where $n' = |X|$.*

Proof. All rotations of X_B (respectively, of \overleftarrow{X}_B) are lexicographically distinct, and no rotation of X_B is lexicographically identical to a rotation of \overleftarrow{X}_B . Thus, we can use Theorem 7 to build the BWT of $\mathcal{R}(X_B) \cup \mathcal{R}(\overleftarrow{X}_B)$ in $O(n'/B)$ time and in $2n'(1+1/k) \log(\sigma+1) + 20n'/B + o(n'/B) \in O(n' \log \sigma)$ bits of working space. Inside the algorithm of Theorem 7, we apply the constant-time operator **right** to the characters of the input BWTs. There is a bijection between set $\mathcal{R}(X_B) \cup \mathcal{R}(\overleftarrow{X}_B)$ and set $\mathcal{R}(X_{B/2})$ that preserves lexicographic order, thus the BWT of $\mathcal{R}(X_{B/2})$ coincides with the BWT of $\mathcal{R}(X_B) \cup \mathcal{R}(\overleftarrow{X}_B)$ in which each character is processed with operator **right**. \square

Lemmas 25 and 26 suggest building the BWT of X in $O(\log B)$ steps, where at step i we compute the BWT of string $X_{B/2^i}$, stopping when $B/2^i = 1$. Note that the key requirement of Lemma 26, i.e. that all rotations of $X_{B/2^i}$ (respectively, of $\overleftarrow{X}_{B/2^i}$) are lexicographically distinct, and that no rotation of $X_{B/2^i}$ is lexicographically identical to a rotation of $\overleftarrow{X}_{B/2^i}$, holds for all i . The time for completing step i is $O(n'/(B/2^i))$, and the Burrows-Wheeler transforms of $X_{B/2^i}$ and of $\overleftarrow{X}_{B/2^i}$ take $O(n' \log \sigma)$ bits of space for every i .

The base case of the recursion is the BWT of string X_B for some initial block size B : we build it using any suffix array construction algorithm that works in $O(\sigma^B + n'/B)$ time and in $O((n'/B) \log(n'/B))$ bits of space (for example those described in [41, 40, 39]). We want this first phase to take $O(n')$ time and $O(n' \log \sigma)$ bits of space, or in other words we want to satisfy the following constraints:

1. $\sigma^B \in O(n')$
2. $(n'/B) \log(n'/B) \in O(n' \log \sigma)$, or more strictly $(n'/B) \log n' \in O(n' \log \sigma)$.

We also want B to be a power of two. Recall that $2^{\lceil \log x \rceil}$ is the smallest power of two that is at least x . Assume thus that we set $B = 2^{\lceil \log(\log n' / (c \log \sigma)) \rceil}$ for some constant c . Then $B \geq \log n' / (c \log \sigma)$, thus Constraint 2 is satisfied by any choice of c . Since $\lceil x \rceil < x + 1$, we have that $B < (2/c) \log n' / \log \sigma$, thus Constraint 1 is satisfied for any $c \geq 2$. For this choice of B the number of steps in the recursion becomes $O(\log \log n')$, and we can read a block of size B in constant time as required by Lemma 26 since the machine word is assumed to be $\Omega(\log n')$. It follows that building the BWT of X takes $O(n' + (n'/B) \sum_{i=1}^{\log B} 2^i) = O(n')$ time and $O(n' \log \sigma)$ bits of working space. Since the BWT of $T\#$ can be derived from the BWT of X at no extra asymptotic cost (see [38]), we have the following result:

Theorem 8. *The BWT of a string $T\#$ such that $T \in [1..\sigma]^n$ and $\# = 0$ can be built in $O(n)$ time and in $O(n \log \sigma)$ bits of working space.*

6 Building string indexes

6.1 Building the compressed suffix array

The *compressed suffix array* of a string $T \in [1..\sigma]^{n-1}\#$ (abbreviated to CSA in what follows) is a representation of SA_T that uses just $O((n \log \sigma)/\epsilon)$ bits space for any given constant ϵ , at the cost of increasing access time to any position of SA_T to $t = O((\log_\sigma n)^\epsilon/\epsilon)$ [31]. Without loss of generality, let B be a block size such that B^i divides n for any setting of i that we will consider, and let T_i be the (suitably terminated) string of length n/B^i defined on the alphabet $[1..(\sigma+1)^{B^i}]$ of all strings of length B^i on alphabet $[0..\sigma]$, and such that the “characters” of T_i correspond to the consecutive blocks of size B^i of T . In other words, $T_i[j] = T[(j-1)B^i + 1..jB^i]$. Note that $T_0 = T$, and T^i with $i > 0$ is the string obtained by grouping every consecutive B characters of T_{i-1} . The CSA of T with parameter ϵ consists of the suffix array of $T_{1/\epsilon}$, and of $1/\epsilon$ layers, where layer $i \in [0..1/\epsilon - 1]$ is composed of the following elements:

1. A data structure that supports **access** and **partialRank** operations⁸ on BWT_{T_i} .
2. The C array of T_i , defined on alphabet $[1..(\sigma+1)^{B^i}]$, encoded as a bitvector with $(\sigma+1)^{B^i}$ ones and n/B^i zeros, and indexed to support **select** queries.
3. A bitvector **marked_i**, of size n/B^i , that marks every position j such that $\text{SA}_{T_i}[j]$ is a multiple of B .

For concreteness, let $\epsilon = 2^{-c}$ for some constant $c > 0$. Note that layer i contains enough information to support function **LF** on T_i . To compute $\text{SA}_{T_i}[j]$, we first check whether **marked_i** $[j] = 1$: if so, then $\text{SA}_{T_i}[j] = B \cdot \text{SA}_{T_{i+1}}[j']$, where $j' = \text{rank}_1(\text{marked}_i, j)$. Otherwise, we iteratively set j to **LF**(j) in constant time and we test whether **marked_i** $[j] = 1$. If it takes t iterations to reach a j^* such that **marked_i** $[j^*] = 1$, then $\text{SA}_{T_i}[j] = B \cdot \text{SA}_{T_{i+1}}[\text{rank}_1(\text{marked}_i, j^*)] + t$. Since $t \leq B - 1$ at any layer, the time spent in a layer is $O(B)$, and the time to traverse all layers is $O(B/\epsilon)$. Setting $B = (\log_\sigma n)^\epsilon$ achieves the claimed time complexity, and assuming without loss of generality that σ is a power of two and $n = \sigma^{2^{2^a}}$ for some integer $a \geq c$ ensures that B^i for any $i \in [1..1/\epsilon]$ is an integer that divides n . Using Lemma 7, every layer takes $O(n \log \sigma)$ bits of space, irrespective of B , so the whole data structure takes $O((n \log \sigma)/\epsilon)$ bits of space. Counting the number **occ** of occurrences of a pattern P in T can be performed in a number of ways with the CSA. A simple, $O(|P| \log n \cdot (\log_\sigma n)^\epsilon/\epsilon)$ time solution, consists in performing binary searches on the suffix array: this allows one to locate all such occurrences in $O(|P|(\log n + \text{occ}) \cdot (\log_\sigma n)^\epsilon/\epsilon)$ time. Alternatively, count queries could be implemented with backward steps as in the BWT index, in overall $O(|P| \log \log \sigma)$ time, using Lemma 11.

The CSA takes in general at least $n \log \sigma + o(n)$ bits, or even $nH_k + o(n)$ bits for $k = o(\log_\sigma n)$ [29]. The CSA has a number of variants, the fastest of which can be built in $O(n \log \log \sigma)$ time using $O(n \log \sigma)$ bits of working space [38]. Combining the setup of data structures described above with Theorem 8 allows one to build the CSA more efficiently:

Theorem 9. *Given a string $T = [1..\sigma]^n$, we can build the compressed suffix array in $O(n)$ time and in $O(n \log \sigma)$ bits of working space.*

Note that the BWT of all strings T_i in the CSA of T , as well as all bitvectors **marked_i**, can be built in a single invocation of Theorem 8, rather than by invoking Theorem 8 $1/\epsilon$ times. Note also that combining Theorem 8 with Lemma 2 and with the first data structure of Lemma 11, yields immediately a BWT index and a succinct suffix array that can be built in deterministic linear time:

Theorem 10. *Given a string $T = [1..\sigma]^n$, we can build the following data structures:*

- A BWT index that takes $n \log \sigma(1 + 1/k) + O(n \log \log \sigma)$ bits of space for any positive integer k , and that implements operation **LF**(i) in constant time for any $i \in [1..n]$, and operation **count**(P) in $O(m(\log \log \sigma + k))$ time for any $P \in [1..\sigma]^m$. The index can be built in $O(n)$ time and in $O(n \log \sigma)$ bits of working space.

⁸The CSA was originally defined in terms of the ψ function: in this case, support for **select** queries would be needed.

- A succinct suffix array that takes $n \log \sigma (1 + 1/k) + O(n \log \log \sigma) + O((n/r) \log n)$ bits of space for any positive integers k and r , and that implements operation **count**(P) in $O(m(\log \log \sigma + k))$ time for any $P \in [1..\sigma]^m$, operation **locate**(i) in $O(r)$ time, and operation **substring**(i, j) in $O(j - i + r)$ time for any $i < j$ in $[1..n]$. The index can be built in $O(n)$ time and in $O(n \log \sigma)$ bits of working space.

6.2 Building BWT indexes

To reduce the time complexity of a backward step to a constant, however, we need to augment the representation of the topology of ST_T described in Lemma 12 with an additional operation, where $T = [1..\sigma]^{n-1}\#$. Recall that the identifier $\text{id}(v)$ of a node v of ST_T is the rank of v in the preorder traversal of ST_T . Given a node v of ST_T and a character $a \in [0..\sigma]$, let operation **weinerLink**($\text{id}(v), a$) return zero if string $a\ell(v)$ is not the prefix of a rotation of T , and return $\text{id}(w)$ otherwise, where w is the locus of string $a\ell(v)$ in ST_T . The following lemma describes how to answer **weinerLink** queries efficiently:

Lemma 27. *Assume that we are given a data structure that supports **access** queries on the BWT of a string $T = [1..\sigma]^{n-1}\#$ in constant time, a data structure that supports **rangeDistinct** queries on BWT_T in constant time per element in the output, and a data structure that supports **select** queries on BWT_T in time t . Assume also that we are given the representation of the topology of ST_T described in Lemma 12. Then, we can build a data structure that takes $O(n \log \log \sigma)$ bits of space and that supports operation **weinerLink**($\text{id}(v), a$) in $O(t)$ time for any node v of ST_T (including leaves) and for any character $a \in [0..\sigma]$. This data structure can be built in randomized $O(nt)$ time and in $O(n \log \sigma)$ bits of working space.*

Proof. We show how to build efficiently the data structure described in [10], which we summarize here for completeness. We use the suffix tree topology to convert in constant time $\text{id}(v)$ to **range**(v) (using operations **leftmostLeaf** and **rightmostLeaf**), and vice versa (using operations **selectLeaf** and **lca**). We traverse ST_T in preorder using the suffix tree topology, as described in Lemma 13. For every internal node v of ST_T , we use a **rangeDistinct** query to compute all the h distinct characters a_1, \dots, a_h that appear in $\text{BWT}_T[\text{range}(v)]$, and for every such character the interval of $a_i\ell(v)$ in BWT_T , in overall $O(h)$ time. Note that the sequence a_1, \dots, a_h returned by a **rangeDistinct** query is not necessarily sorted in lexicographic order. We determine whether $a_i\ell(v)$ is the label of a node w of ST_T by taking a suffix link from the locus of $a_i\ell(v)$ in $O(t)$ time, using Lemma 15, and by checking whether the destination of such link is indeed v .

For every character $c \in [0..\sigma]$, we use vector **sources** ^{c} to store all nodes v of ST_T (including leaves) that are the source of an implicit or explicit Weiner link labeled by c , in the order induced by the preorder traversal of ST_T . We encode the difference between the preorder ranks of two consecutive nodes in the same **sources** ^{c} using Elias delta or gamma coding [19]. We also store a bitvector **explicit** ^{c} that marks with a one every explicit Weiner link in **sources** ^{c} (recall that Weiner links from leaves are explicit). Bitvectors **sources** ^{c} and **explicit** ^{c} can be filled during the preorder traversal of ST_T . Once **explicit** ^{c} has been filled, we index it to answer **rank** queries. The space used by such indexed bivectors **explicit** ^{c} for all $c \in [0..\sigma]$ is $O(n)$ bits by Observation 1, and the space used by vectors **sources** ^{c} for all $c \in [0..\sigma]$ is $O(n \log \sigma)$ bits, by applying Jensen's inequality twice as in Lemma 20. We follow the static allocation strategy described in Section 3.1: specifically, we compute the number of bits needed by **sources** ^{c} and **explicit** ^{c} during a preliminary pass over ST_T , in which we increment the size of the arrays by keeping the preorder position of the last internal node with a Weiner link labeled by c , for all $c \in [0..\sigma]$. This preprocessing takes $O(n)$ time and $O(\sigma \log n) \in o(n)$ bits of space. Once such sizes are known, we allocate a large enough contiguous region of memory.

Finally, we build an array $C'[1..\sigma]$ where $C'[a]$ is the number of nodes v in ST_T (including leaves) such that $\ell(v)$ starts with a character strictly smaller than a . We also build an implementation of an MMPHF f^c for every **sources** ^{c} using the technique described in the proof of Lemma 20, and we discard **sources** ^{c} . All such MMPHF implementations take $O(n \log \log \sigma)$ bits of space, and they can be built in overall $O(n)$ randomized time and in $O(\sigma^k \log \sigma)$ bits of working space, for any integer $k > 1$. Note that C' takes $O(\sigma \log n) \in o(n)$ bits of space, since $\sigma \in o(\sqrt{n}/\log n)$, and it can be built with a linear-time preorder traversal of ST_T .

Given a node v of ST_T and a character $c \in [0..\sigma]$, we determine whether the Weiner link from v labeled by c is explicit or implicit by accessing **explicit** ^{c} ($f^c(\text{id}(v))$), and we compute the identifier of the locus w

of the destination of the Weiner link (which might be a leaf) by computing:

$$C'[c] + \text{rank}_1(\text{explicit}^c, f^c(\text{id}(v)) - 1) + 1$$

If there is no Weiner link from v labeled by c , then v does not belong to sources^c , but $f^c(\text{id}(v))$ still returns a valid pointer in sources^c : to check whether this pointer corresponds to v , we convert v and w to intervals in BWT_T using the suffix tree topology, and we check whether $\text{select}_c(\text{BWT}_T, \text{sp}(w) - C[c]) \in \text{range}(v)$.

The output of this construction consists in arrays C' , explicit^c , and in the implementation of f^c , for all $c \in [0..\sigma]$. \square

Since operation $\text{weinerLink}(\text{id}(v), a)$ coincides with a backward step with character a from $\text{range}(v)$ in BWT_T , Lemma 27 enables the construction of space-efficient BWT indexes with constant-time LF:

Theorem 11. *Given a string $T = [1..\sigma]^{n-1}\#$, we can build any of the following data structures in randomized $O(n)$ time and in $O(n \log \sigma)$ bits of working space:*

- *A BWT index that takes $n \log \sigma(1 + 1/k) + O(n \log \log \sigma)$ bits of space for any positive integer k , and that implements operation $\text{LF}(i)$ in constant time for any $i \in [1..n]$, and operation $\text{count}(P)$ in $O(mk)$ time for any $P \in [1..\sigma]^m$.*
- *A succinct suffix array that takes $n \log \sigma(1 + 1/k) + O(n \log \log \sigma) + O((n/r) \log n)$ bits of space for any positive integers k and r , and that implements operation $\text{count}(P)$ in $O(mk)$ time for any $P \in [1..\sigma]^m$, operation $\text{locate}(i)$ in $O(r)$ time, and operation $\text{substring}(i, j)$ in $O(j - i + r)$ time for any $i < j$ in $[1..n]$.*

Alternatively, for the same construction space and time, we can build analogous data structures that support $\text{LF}(i)$ in $O(k)$ time, $\text{count}(P)$ in $O(m)$ time, $\text{locate}(i)$ in $O(r)$ time, and $\text{substring}(i, j)$ in $O(j - i + r)$ time: such data structures take the same space as those described above.

Proof. In this proof we combine a number of results described earlier in the paper: see Figure 1 for a summary of their mutual dependencies.

We use Theorem 8 to build BWT_T from T , and Lemma 7 to build a data structure that supports **access**, **partialRank** and **select** queries on BWT_T . Then, we discard BWT_T . Together with the C array of T , this is already enough to implement function LF and to build arrays **samples** and **pos2rank** for the succinct suffix array, using Lemma 2. We either use the data structure of Lemma 7 that supports select queries in $O(k)$ time (in which case we implement locate and substring queries with function LF), or the data structure that supports select queries in constant time (in which case we implement locate and substring queries with function ψ).

To implement backward steps we need support for **weinerLink** operations from internal nodes of ST_T . We use Lemma 19 to build a **rangeDistinct** data structure on BWT_T from the **access**, **partialRank** and **select** data structure built by Lemma 7. We use **rangeDistinct** queries inside the algorithm to enumerate the BWT intervals of all internal nodes of ST_T described in Theorem 3, and we use such algorithm to build the balanced parentheses representation of ST_T as described in Theorem 4. To support operations on the topology of ST_T , we feed the balanced parentheses representation of ST_T to Lemma 12. Finally, we use the **rangeDistinct** data structure, the tree topology, and the support for **access**, **partialRank** and **select** queries on BWT_T , to build the data structures that support **weinerLink** operations described in Lemma 27. At the end of this process, we discard the **rangeDistinct** data structure.

The output of this construction consists of the data structures that support **access**, **partialRank**, and **select** on BWT_T , and **weinerLink** on ST_T . \square

6.3 Building the bidirectional BWT index

The BWT index can be made *bidirectional*, in the sense that it can be adapted to support both left and right extension by a single character [65, 66]. In addition to having a number of applications in high-throughput

sequencing (see e.g. [43, 44]), this index can be used to implement a number of string analysis algorithms, and as an intermediate step for building the compressed suffix tree.

Given a string $T = t_1 t_2 \dots t_{n-1}$ on alphabet $[1..\sigma]$, consider two BWT transforms, one built on $T\#$ and one built on $\underline{T}\# = t_n t_{n-1} \dots t_1 \#$. Let $\mathbb{I}(W, T)$ be the function that returns the interval in $\text{BWT}_{T\#}$ of the suffixes of $T\#$ that are prefixed by string $W \in [1..\sigma]^+$. Note that interval $\mathbb{I}(W, T)$ in the *suffix array* of $T\#$ contains all the starting positions of string W in T . Symmetrically, interval $\mathbb{I}(\underline{W}, \underline{T})$ in the suffix array of $\underline{T}\#$ contains all those positions i such that $n - i + 1$ is an *ending position* of string W in T .

Definition 7. *Given a string $T \in [1..\sigma]^{n-1}$, a bidirectional BWT index on T is a data structure that supports the following operations on pairs of integers $1 \leq i \leq j \leq n$ and on substrings W of T :*

- **isLeftMaximal**(i, j): *returns 1 if substring $\text{BWT}_{T\#}[i..j]$ contains at least two distinct characters, and 0 otherwise.*
- **isRightMaximal**(i, j): *returns 1 if substring $\text{BWT}_{\underline{T}\#}[i..j]$ contains at least two distinct characters, and 0 otherwise.*
- **enumerateLeft**(i, j): *returns all the distinct characters that appear in substring $\text{BWT}_{T\#}[i..j]$, in lexicographic order.*
- **enumerateRight**(i, j): *returns all the distinct characters that appear in $\text{BWT}_{\underline{T}\#}[i..j]$, in lexicographic order.*
- **extendLeft**($c, \mathbb{I}(W, T), \mathbb{I}(\underline{W}, \underline{T})$): *returns pair $(\mathbb{I}(cW, T), \mathbb{I}(\underline{W}c, \underline{T}))$ for $c \in [0..\sigma]$.*
- **extendRight**($c, \mathbb{I}(W, T), \mathbb{I}(\underline{W}, \underline{T})$): *returns $(\mathbb{I}(Wc, T), \mathbb{I}(c\underline{W}, \underline{T}))$ for $c \in [0..\sigma]$.*
- **contractLeft**($\mathbb{I}(aW, T), \mathbb{I}(\underline{W}a, \underline{T})$), *where $a \in [1..\sigma]$ and aW is right-maximal: returns pair $(\mathbb{I}(W, T), \mathbb{I}(\underline{W}, \underline{T}))$;*
- **contractRight**($\mathbb{I}(Wb, T), \mathbb{I}(b\underline{W}, \underline{T})$), *where $b \in [1..\sigma]$ and Wb is left-maximal: returns pair $(\mathbb{I}(W, T), \mathbb{I}(\underline{W}, \underline{T}))$.*

Operations **extendLeft** and **extendRight** are analogous to a standard backward step in $\text{BWT}_{T\#}$ or $\text{BWT}_{\underline{T}\#}$, but they keep the interval of a string W in one BWT *synchronized* with the interval of its reverse \underline{W} in the other BWT.

In order to build a bidirectional BWT index on string T , we also need to support operation **countSmaller**(**range**(v), c), which returns the number of occurrences of characters smaller than c in $\text{BWT}_{T\#}[\text{range}(v)]$, where v is a node of $\text{ST}_{T\#}$ and c is the label of an explicit or implicit Weiner link from v . Note that, when v is a leaf of $\text{ST}_{T\#}$, **countSmaller**(**range**(v), $\text{BWT}_{T\#}[x]$) = 0, where $x = \text{sp}(v) = \text{ep}(v)$. The construction of Lemma 27 can be extended to support constant-time **countSmaller** queries, as described in the following lemma:

Lemma 28. *Assume that we are given a data structure that supports **access** queries on the BWT of a string $T = [1..\sigma]^{n-1}\#$ in constant time, a data structure that supports **rangeDistinct** queries on BWT_T in constant time per element in the output, and a data structure that supports **select** queries on BWT_T in time t . Assume also that we are given the representation of the topology of ST_T described in Lemma 12. Then, we can build a data structure that takes $3n \log \sigma + O(n \log \log \sigma)$ bits of space, and that supports operation **weinerLink**(**id**(v), a) in $O(t)$ time for any node v of ST_T (including leaves) and for any character $a \in [0..\sigma]$, and operation **countSmaller**(**range**(v), c) in constant time for any internal node v of ST_T and for any character $a \in [1..\sigma]$ that labels a Weiner link from v . This data structure can be built in randomized $O(nt)$ time and in $O(n \log \sigma)$ bits of working space.*

Proof. We run the algorithm described in the proof of Lemma 27. Specifically, we traverse ST_T in preorder, we print arrays **sources** ^{c} for all $c \in [0..\sigma]$, and we build the implementation of an MMPHF f^c for every **sources** ^{c} . Before discarding **sources** ^{c} , we build the prefix-sum data structure of Lemma 4 on every **sources** ^{c} with $c > 0$: by Jensen's inequality and Observation 1, all such data structures take at most $3n \log \sigma + 6n + o(n)$ bits of space in total.

Let $ST^c = (V^c, E^c)$ be the contraction of ST_T induced by all the n^c nodes (including leaves) that have an explicit or implicit Weiner link labeled by character $c \in [1..\sigma]$. During the preorder traversal of ST_T , we also concatenate to a bitvector **parentheses**^c an open parenthesis every time we visit an internal node v with a Weiner link labeled by character c from its parent, and a closed parenthesis every time we visit v from its last child. Note that **parentheses**^c represents the topology of ST^c . By Observation 1, building all bitvectors **parentheses**^c takes $O(n)$ time and $6n + o(n)$ bits of space in total, since every pair of corresponding parentheses can be charged to an explicit or implicit Weiner link of ST_T . We feed **parentheses**^c to Lemma 12 to obtain support for tree operations, and we discard **parentheses**^c. Following the strategy described in Section 3.1, we preallocate the space required by **sources**^c and **parentheses**^c for all $c \in [1..\sigma]$ during a preliminary pass over ST_T . Note that the preorder rank in ST^c of a node v , that we denote by $\text{id}^c(v)$, equals its position in array **sources**^c. Note also that the set of $\text{id}^c(w)$ values for all the descendants w of v in ST^c , including v itself, forms a contiguous range.

We allocate σ empty arrays **diff**^c[1.. n^c] which, at the end of the algorithm, will contain the following information:

$$\begin{aligned} \text{diff}^c[\text{id}^c(v)] &= \text{countSmaller}(\text{range}(v), c) \\ &\quad - \sum_{(v,w) \in E_c} \text{countSmaller}(\text{range}(w), c) \end{aligned}$$

i.e. **diff**^c[k] will encode the difference between the number of characters smaller than c in the BWT interval of the node v of ST_T that is mapped to position k in **sources**^c, and the number of characters smaller than c in the BWT intervals of all the descendants of v in the contracted suffix tree ST^c . To compute $\text{countSmaller}(\text{range}(v), c)$ for some internal node v of ST_T , we proceed as follows. We use the implementation of the MMPHF f^c built on **sources**^c to compute $\text{id}^c(v)$, we retrieve the smallest and the largest $\text{id}^c(w)$ value assumed by a descendant w of v in ST^c using operations **leftmostLeaf** and **rightmostLeaf** provided by the topology of ST^c , and we sum **diff**^c[k] for all k in this range. We compute this sum in constant time by encoding **diff**^c with the prefix-sum data structure described in Lemma 4. Since $\sum_{k=1}^{n^c} \text{diff}^c[k] \leq n$, the total space taken by all such prefix-sum data structures is at most $3n \log \sigma + 6n + o(n)$ bits, by Observation 1 and Jensen's inequality.

To build the **diff**^c arrays, we scan the sequence of all characters $c_1 < c_2 < \dots < c_k$ such that $c_i \in [1..\sigma]$ and ST^{c_i} has at least one node, for all $i \in [1..k]$. We use a temporary vector **lastChar** with one element per node of ST_T : after having processed character c_i , **lastChar**[$\text{id}(v)$] stores the largest $c_j \leq c_i$ that labels a Weiner link from v . We also assume to be able to answer $\text{countSmaller}(\text{range}(v), c)$ queries in constant time. Note that **lastChar** takes at most $(2n - 1) \log \sigma$ bits of space. We process character c_i as follows. We traverse ST^{c_i} in preorder using its topology, as described in Lemma 13. For each node v of ST^{c_i} , we use $\text{id}^{c_i}(v)$ and the prefix-sum data structure on **sources** ^{c_i} to compute $\text{id}(v)$. If v is an internal node of ST_T , we use $\text{id}(v)$ to access $b = \text{lastChar}[\text{id}(v)]$. We compute the number of occurrences of character b in $\text{range}(v)$ using the $O(t)$ -time operation **weinerLink**($\text{id}(v), b$), and we compute the number of occurrences of characters smaller than b in $\text{range}(v)$ using the constant-time operation $\text{countSmaller}(\text{range}(v), b)$. We do the same for all children of v in ST^{c_i} , which we can access using the topology of ST^{c_i} . Finally, we sum the values of all children and we subtract this sum from the value of v , appending the result to the end of **diff** ^{c_i} using Elias delta or gamma coding. Finally, we set **lastChar**[$\text{id}(v)$] = c_i . The total number of accesses to a node v of ST_T is a constant multiplied by the number of Weiner links from v , thus the algorithm runs in $O(nt)$ time.

The output of the construction consists in the topology of ST^{c_i} for all $i \in [1..k]$, in arrays C' and **explicit**^c of Lemma 27 for all $c \in [0..\sigma]$, in the implementation of f^c for all $c \in [0..\sigma]$, and in the prefix-sum data structure on **diff** ^{c_i} for all $i \in [1..k]$. \square

Lemma 28 immediately yields the following result:

Theorem 12. *Given a string $T = [1..\sigma]^n$, we can build in randomized $O(n)$ time and in $O(n \log \sigma)$ bits of working space a bidirectional BWT index that takes $O(n \log \sigma)$ bits of space and that implements every operation in time linear in the size of its output.*

Proof. Let W be a substring of T such that $\mathbb{I}(W, \text{BWT}_{T\#}) = [i..j]$ and $\mathbb{I}(\underline{W}, \text{BWT}_{\underline{T}\#}) = [i'..j']$, let v be the node of $\text{ST}_{T\#}$ such that $\mathbb{I}(v, \text{BWT}_{T\#}) = [i..j]$ and let v' be the node of $\text{ST}_{\underline{T}\#}$ such that $\mathbb{I}(v', \text{BWT}_{\underline{T}\#}) = [i'..j']$. We plug the `countSmaller` support provided by Lemma 28 in the construction of the BWT index described in Theorem 11, and we build the corresponding data structures on both $\text{BWT}_{T\#}$ and $\text{BWT}_{\underline{T}\#}$.

Operation `extendLeft` $(a, (i, j), (i', j')) = ((p, q), (p', q'))$ can be implemented as follows: we compute (p, q) using `weinerLink` $(\text{id}(v), a)$, and we set $(p', q') = (i' + \text{countSmaller}(i, j, a), i' + \text{countSmaller}(i, j, a) + q - p)$.

To support `isLeftMaximal` we build a bitvector `runs` $[2..n + 1]$ such that `runs` $[i] = 1$ if and only if $\text{BWT}_{T\#}[i] \neq \text{BWT}_{T\#}[i - 1]$. We build this vector by a linear scan of $\text{BWT}_{T\#}$, and we index it to support `rank` queries in constant time. We implement `isLeftMaximal` (i, j) by checking whether there is a one in `runs` $[i + 1..j]$, i.e. whether $\text{rank}_1(\text{runs}, j) - \text{rank}_1(\text{runs}, i) \geq 1$. This technique was already described in e.g. [42, 56].

Assuming that W is right-maximal, we support `contractLeft` $((i, j), (i', j')) = ((p, q), (p', q'))$ as follows. Let $W = aV$ for some $a \in [0..\sigma]$ and $V \in [1..\sigma]^*$. We compute $(p, q) = \mathbb{I}(V, \text{BWT}_{T\#})$ using operation `suffixLink` $(\text{id}(v))$ described in Lemma 15, and we check the result of operation `isLeftMaximal` (p, q) : if V is not left-maximal, then $(p', q') = (i', j')$, otherwise \underline{V} is the label of an internal node of $\text{ST}_{\underline{T}\#}$, and this node is the parent of v' .

To implement `enumerateLeft` (i, j) , we first check whether `isLeftMaximal` (i, j) returns true: otherwise, there is just character $\text{BWT}_{T\#}[i]$ to the left of W in $T\#$. Recall that operation `rangeDistinct` (i, j) on $\text{BWT}_{T\#}$ returns the distinct characters that occur in $\text{BWT}_{T\#}[i..j]$ as a sequence a_1, \dots, a_h which is not necessarily sorted lexicographically. Note that characters a_1, \dots, a_h are precisely the distinct right-extensions of string \underline{W} in $\underline{T}\#$: since W is left-maximal, we have that $\underline{W} = \ell(v')$, and a_1, \dots, a_h are the labels associated with the children of v' in $\text{ST}_{\underline{T}\#}$. Thus, if we had an MMPHF $f^{v'}$ that maps a_1, \dots, a_h to their rank among the children of v' in $\text{ST}_{\underline{T}\#}$, we could sort the output of `rangeDistinct` (i, j) in linear time. We can build the implementation of $f^{v'}$ for all internal nodes v' of $\text{ST}_{\underline{T}\#}$ using the enumeration algorithm described in Theorem 3, and by applying to array `chars` of `repr` $(\ell(v'))$ the implementation of the MMPHF described in Lemma 16. Since every character in every `chars` array can be charged to a distinct node of $\text{ST}_{\underline{T}\#}$, the set of all such MMPHF implementations takes $O(n \log \log \sigma)$ bits of space, and building it takes randomized $O(n)$ time and $O(\sigma \log \sigma)$ bits of working space. Operation `enumerateLeft` can be combined with `extendLeft` to return intervals in addition to distinct characters.

We support `enumerateRight`, `isRightMaximal`, `contractRight` and `extendRight` symmetrically. \square

Before describing the construction of other indexes, we note that the constant-time `countSmaller` support of Lemma 28, combined with the enumeration algorithm of Lemma 22, enables an efficient way of building $\text{BWT}_{\underline{T}\#}$ from $\text{BWT}_{T\#}$:

Lemma 29. *Let $T \in [1..\sigma]^n$ be a string. Given $\text{BWT}_{T\#}$, indexed to support `rangeDistinct` queries in constant time per element in their output, and `countSmaller` queries in constant time, we can build $\text{BWT}_{\underline{T}\#}$ in $O(n)$ time and in $O(\sigma^2 \log^2 n)$ bits of working space, and we can build $\text{BWT}_{\underline{T}\#}$ from left to right, in $O(n)$ time and in $O(\lambda_T \cdot \sigma^2 \log n)$ bits of working space, where λ_T is defined in Section 2.2.*

Proof. We use Lemma 22 to iterate over all right-maximal substrings W of T , and we use `countSmaller` queries to keep at every step, in addition to `repr` (W) , the interval of \underline{W} in $\text{BWT}_{\underline{T}\#}$, as described in Theorem 12.

Let $a \in [1..\sigma]$, let $\mathbb{I}(\underline{W}, \text{BWT}_{\underline{T}\#}) = [i..j]$, and let $\mathbb{I}(a\underline{W}, \text{BWT}_{\underline{T}\#}) = [i'..j']$. Recall that $[i'..j'] \subseteq [i..j]$, and that we can test whether $a\underline{W}$ is right-maximal by checking whether `gamma` $[a] > 1$ in Lemma 21. If $a\underline{W}$ is not right-maximal, i.e. if the Weiner link labelled by a from the locus of W in $\text{ST}_{T\#}$ is implicit, then $\text{BWT}_{\underline{T}\#}[i'..j']$ is a run of character $A[a][1]$, where A is the matrix used in Lemma 21. If $a\underline{W}$ is right-maximal, then it will be processed in the same way as W during the iteration, and its corresponding interval $[i'..j']$ in $\text{BWT}_{\underline{T}\#}$ will be recursively filled.

To build $\text{BWT}_{\underline{T}\#}$ from left to right, it suffices to replace the traversal strategy of Lemma 22, based on the logarithmic stack technique, with a traversal based on the lexicographic order of the left-extensions of every right-maximal substring. This makes the depth of the traversal stack of Lemma 22 become $O(\lambda_T)$. \square

Contrary to the algorithm described in [53], Lemma 29 does not need T and $\text{SA}_{T\#}$ in addition to $\text{BWT}_{T\#}$.

We also note that a fast bidirectional BWT index, such as the one in Theorem 12, enables a number of applications, which we will describe in more detail in Section 7. For example, we can enumerate all the right-maximal substrings of T as in Section 4, but with the additional advantage of providing access to their left extensions in lexicographic order:

Lemma 30. *Given the bidirectional BWT index of $T \in [1..\sigma]^n$ described in Theorem 12, there is an algorithm that solves Problem 2 in $O(n)$ time, and in $O(\sigma \log^2 n)$ bits of working space and $O(\sigma^2 \log n)$ bits of temporary space, where the sequence a_1, \dots, a_h of left-extensions of every right-maximal string W is in lexicographic order.*

Proof. By adapting Lemma 22 to use operations `enumerateLeft`, `extendLeft` and `isRightMaximal` provided by the bidirectional BWT index. The smaller working space with respect to Lemma 22 derives from the fact that the representation of a string W is now the constant-space pair of intervals $(\mathbb{I}(W, T\#), \mathbb{I}(W, \underline{T}\#))$. \square

6.4 Building the permuted LCP array

We can use the bidirectional BWT index to compute the permuted LCP array as well:

Lemma 31. *Given the bidirectional BWT index of $T \in [1..\sigma]^n$ described in Theorem 12, we can build $\text{PLCP}_{T\#}$ in $O(n)$ time and in $O(\log n)$ bits of working space.*

Proof. We scan $T' = T\#$ from left to right. By inverting $\text{BWT}_{\underline{T}\#}$, we know in constant time the position r_i in $\text{BWT}_{T\#}$ that corresponds to every position i in $T\#$. Assume that we know $\text{PLCP}[i]$ and the interval of $aW = T[i..i + \text{PLCP}[i] - 1]$ in $\text{BWT}_{T\#}$ and in $\text{BWT}_{\underline{T}\#}$, where $a \in [1..\sigma]$. Note that aW is right-maximal, thus we can take the suffix link from the internal node of the suffix tree of $T\#$ labeled by aW to the internal node labeled by W , using operation `contractLeft`. Let $([x..y], [x'..y'])$ be the intervals of W in $\text{BWT}_{T\#}$ and in $\text{BWT}_{\underline{T}\#}$, respectively. If $i = 0$ or $\text{PLCP}[i] = 0$, rather than taking the suffix link from aW , we set $W = \varepsilon$, $x = x' = 1$ and $y = y' = n + 1$. Since $\text{PLCP}[i + 1] \geq \text{PLCP}[i] - 1$, we set $\text{PLCP}[i + 1]$ to its lower bound $|W|$. Then, we issue:

$$([x..y], [x'..y']) \leftarrow \text{extendRight}(T'[i + \text{PLCP}[i]], [x..y], [x'..y'])$$

and we check whether $x = r_{i+1}$: if this is the case we stop, since neither $W \cdot T'[i + \text{PLCP}[i]]$ nor any of its right-extensions are prefixes of the suffix at position $r_{i+1} - 1$ in $\text{BWT}_{T\#}$. Otherwise, we increment $\text{PLCP}[i + 1]$ by one and we continue issuing `extendRight` operations with the following character of T' . At the end of this process we know the interval of $T'[i + 1..i + 1 + \text{PLCP}[i + 1] - 1]$ in $\text{BWT}_{T\#}$ and $\text{BWT}_{\underline{T}\#}$, thus we can repeat the algorithm from position $i + 2$. \square

This algorithm can be easily adapted to compute the *distinguishing statistics array* of a string T given its bidirectional BWT index, and to compute the *matching statistics array* of a string T^2 with respect to a string T^1 , given the bidirectional BWT index of $T^1\#_1T^2\#_2$: see Section 7.1.

6.5 Building the compressed suffix tree

The *compressed suffix tree* of a string $T \in [1..\sigma]^{n-1}$ [63], abbreviated to CST in what follows, is an index that consists of the following elements:

1. The compressed suffix array of $T\#$.
2. The topology of the suffix tree of $T\#$. This takes $4n + o(n)$ bits of space, but it can be reduced to $2.54n + o(n)$ bits [26].
3. The permuted LCP array of $T\#$, which takes $2n + o(n)$ bits of space [63].

The CST is designed to support the same set of operations as the suffix tree. Specifically, all operations that involve just the suffix tree topology can be supported in constant time, including taking the parent of a node and the lowest common ancestor of two nodes. Most of the remaining operations are instead supported in time t , i.e. in the time required for accessing the value stored at a given suffix array position. Some operations are supported by augmenting the CST with other data structures: for example, following the edge that connects a node to its child with a given label (and returning an error if no such edge exists) needs additional $O(n \log \log \sigma)$ bits, and runs in t time. Some operations take even more time: for example, string level ancestor queries (defined in Section 6.5) need additional $o(n)$ bits of space, and are supported in $O(t \log \log n)$ time.

By just combining Lemma 31 with Theorems 12, 9, 4 and 8, we can prove the key result of Section 6:

Theorem 13. *Given a string $T = [1..\sigma]^n$, we can build the three main components of the compressed suffix tree (i.e. the compressed suffix array, the suffix tree topology, and the permuted LCP array) in randomized $O(n)$ time and in $O(n \log \sigma)$ bits of working space. Such components take overall $O(n \log \sigma)$ bits of space.*

A number of applications of the suffix tree depend on the following additional operations: **stringDepth**($\text{id}(v)$), which returns the length of the label of a node v of the suffix tree; **blindChild**($\text{id}(v), a$), which returns the identifier of the child w of a node v of the suffix tree such that $\ell(v, w) = aW$ for some $W \in \Sigma^*$ and $a \in \Sigma$, and whose output is undefined if v has no outgoing edge whose label starts with a ; **child**($\text{id}(v), a$), which is analogous to **blindChild** but returns \emptyset if v has no outgoing edge whose label starts with a ; and **stringAncestor**($\text{id}(v), d$), which returns the locus of the prefix of length d of $\ell(v)$. The latter operation is called *string level ancestor query*. Operation **stringDepth** can be supported in $O((\log_\sigma^\epsilon n)/\epsilon)$ time using just the three main components of the compressed suffix tree. To support **blindChild** and **child** we need the following additional structure:

Lemma 32. *Given a string $T = [1..\sigma]^n$, we can build in randomized $O(n)$ time and in $O(n \log \sigma)$ bits of working space, a data structure that allows a compressed suffix tree to support operation **blindChild** in constant time, and operation **child** in $O((\log_\sigma^\epsilon n)/\epsilon)$ time. Such data structure takes $O(n \log \log \sigma)$ bits of space.*

Proof. We build the following data structures, described in [5, 10]. We use an array **nChildren**[$1..2n-1$], of $(2n-1) \log \sigma$ bits, to store the number of children of every suffix tree node, in preorder, and we use an array **labels**[$1..2n-2$] of $(2n-2) \log \sigma$ bits to store the sorted labels of the children of every node, in preorder. We enumerate the BWT intervals of every right-maximal substring W of T , as well as the number k of distinct right-extensions of W , using Theorem 3. We convert **range**(W) into the preorder identifier i of the corresponding suffix tree node using the tree topology, and we set **nChildren**[i] = k . Then, we build the prefix-sum data structure of Lemma 4 on array **nChildren** (recall that this structure takes $O(n)$ bits of space), and we enumerate again the BWT interval of every right-maximal substring W of T , along with its right-extensions b_1, b_2, \dots, b_k , using Theorem 3. For every such W , we set **labels**[$i+j$] = b_j for all $j \in [0..k-1]$, where i is computed from the prefix-sum data structure. Finally, we scan **nChildren** and **labels** using pointers i and j , respectively, both initialized to one, we iteratively build a monotone minimal perfect hash function on **labels**[$j..j + \text{nChildren}[i] - 1$] using Lemma 17, and we set i to $i+1$ and j to $j + \text{nChildren}[i]$. All such MMPHF fit in $O(n \log \log \sigma)$ bits of space and they can be built in randomized $O(n)$ time. \square

The output of **blindChild** can be checked in $O((\log_\sigma^\epsilon n)/\epsilon)$ time using the compressed suffix array and the **stringDepth** operation, assuming we store the original string. Finally, the data structures that support string level ancestor queries can be built in deterministic linear time and in $O(n \log \sigma)$ bits of space:

Lemma 33. *Given the compressed suffix tree of a string $T = [1..\sigma]^n$, we can build in $O(n)$ time and in $O(n \log \sigma)$ bits of working space, a data structure that allows the compressed suffix tree to answer **stringAncestor** queries in $O(((\log_\sigma^\epsilon n)/\epsilon) \log^\epsilon n \cdot \log \log n)$ time. Such data structure takes $o(n)$ bits of space.*

Proof. We call *depth* of a node the number of edges in the path that connects it to the root, and we call *height* of an internal node v the difference between the depth of the deepest leaf in the subtree rooted at v and the

depth of v . To build the data structure, we first sample a node every b in the suffix tree. Specifically, we sample a node iff its depth is multiple of b and its height is at least b . Note that the number of sampled nodes is at most n/b , since we can associate at least $b - 1$ non-sampled nodes to every sampled node. Specifically, let v be a sampled node at depth ib for some i . If no descendant of v is sampled, we can assign to v all the at least b nodes in the path from v to its deepest leaf. If at least one descendant of v is sampled, then v has at least one sampled descendant w at depth $(i + 1)b$, and we can assign to v all the $b - 1$ non-sampled nodes in the path from v to w .

We perform the sampling using just operations supported by the balanced parentheses representation of the topology of the suffix tree (see Lemma 12). Specifically, we perform a preorder traversal of the suffix tree topology using Lemma 13, we compute the depth and the height of every node v using operations **depth** and **height** provided by the balanced parentheses representation, and, if v has to be sampled, we append pair $(\text{id}(v), \text{stringDepth}(\text{id}(v)))$ to a temporary list **pairs**. Building **pairs** takes $O((n/b) \cdot (\log_\sigma^\epsilon n)/\epsilon)$ time, and **pairs** itself takes $O((n/b) \log n)$ bits of space. During the traversal we also build a sequence of balanced parentheses S , that encodes the topology of the subgraph of the suffix tree induced by sampled nodes: every time we traverse a sampled node from its parent we append to S an opening parenthesis, and every time we traverse a sampled node from its last child we append to S a closing parentheses. At the end of this process, we build a weighted level ancestor data structure (WLA, see e.g. [1]) on the set of sampled nodes, where the weight assigned to a node equals its string depth. To do so, we build the data structure of Lemma 12 on S , and we feed S and **pairs** to the algorithm described in [1]. The WLA data structure takes $O(n/b)$ space and it can be built in $O(n/b)$ time. Finally, we build a dictionary D that stores the identifiers of all sampled nodes: the size of this dictionary is $O((n/b) \log n)$ bits. The dictionary and the WLA data structure are the output of our construction.

We now describe how to answer **stringAncestor**($\text{id}(v), d$), waving details on corner cases for brevity. We first check whether w , the lowest ancestor of v at depth ib for some i , is sampled: to do so, we compute $e = \text{depth}(\text{id}(v))$, we issue **ancestor**($\text{id}(v), ib$), where $i = \lfloor e/b \rfloor$, using the suffix tree topology, and we query D with $\text{id}(w)$. If w is not sampled, we replace w with its ancestor at depth $(i - 1)b$, which is necessarily sampled. If the string depth of w equals d , we return $\text{id}(w)$. Otherwise, if the string depth of w is less than d , we perform a binary search over the range of tree depths between the depth of w plus one and the depth of v , using operations **ancestor** and **stringDepth**. Otherwise, we query the WLA data structure to determine u , the deepest sampled ancestor of w whose depth is less than d , and we perform a binary search over the range of depths between the depth of u plus one and the depth of w , using operations **ancestor** and **stringDepth**. Note that the range explored by the binary search is of size at most $2b$, thus the search takes $O(\log b)$ steps and $O(\log b \cdot ((\log_\sigma^\epsilon n)/\epsilon))$ time. Setting $b = \log^2 n$ makes the query time $O(\log \log n \cdot ((\log_\sigma^\epsilon n)/\epsilon))$, the time to build **pairs** $O(n)$, and the space taken by **pairs** and by the WLA data structure $o(n)$ bits. \square

7 String analysis

In this section we use the enumerators of right-maximal substrings described in Theorems 3 and 6 to solve a number of fundamental string analysis problems in optimal deterministic time and small space. Specifically, we show that all such problems can be solved efficiently by just implementing function `callback` invoked by Algorithms 2 and 3. We also show how to compute *matching statistics* and *distinguishing statistics* (defined below) using a bidirectional BWT index.

7.1 Matching statistics

Definition 8 ([68, 67]). *Given two strings $S \in [1..\sigma]^n$ and $T \in [1..\sigma]^m$, and an integer threshold $\tau > 0$, the matching statistics $\text{MS}_{T,S,\tau}$ of T with respect to S is a vector of length m that stores at index $i \in [1..m]$ the length of the longest prefix of $T[i..m]$ that occurs at least τ times in S .*

Definition 9 ([68, 67]). *Given $S \in [1..\sigma]^n$ and an integer threshold $\tau > 0$, the distinguishing statistics $\text{DS}_{S,\tau}$ of S is a vector of length $|S|$ that stores at index $i \in [1..|S|]$ the length of the shortest prefix of $S[i..|S|]$ that occurs at most τ times in S .*

We drop a subscript from $\text{DS}_{S,\tau}$ whenever it is clear from the context. Note that $\text{DS}_{S,\tau}[i] \geq 1$ for all i and τ . The key additional property of $\text{DS}_{S,\tau}$, which is shared by $\text{PLCP}_{S\#}$, is called δ -monotonicity:

Definition 10 ([59]). *Let $a = a_0a_1 \dots a_n$ and $\delta = \delta_1\delta_2 \dots \delta_n$ be two sequences of nonnegative integers. Sequence a is said to be δ -monotone if $a_i - a_{i-1} \geq -\delta_i$ for all $i \in [1..n]$.*

Specifically, $\text{MS}_{T,S,\tau}[i] - \text{MS}_{T,S,\tau}[i-1] \geq -1$ for all $i \in [2..m]$, $\text{DS}_{T,\tau}[i] - \text{DS}_{T,\tau}[i-1] \geq -1$ and $\text{PLCP}_{T\#}[i] - \text{PLCP}_{T\#}[i-1] \geq -1$ for all $i \in [2..m+1]$. This property allows all three of these vectors to be encoded in $2x$ bits, where x is the length of the corresponding input string [62, 8].

The matching statistics array and the distinguishing statistics array of a string can be built in linear time from the bidirectional BWT index of Theorem 12:

Lemma 34. *Given a bidirectional BWT index of $T \in [1..\sigma]^n$ that supports every operation in time linear in the size of its output, we can build $\text{DS}_{T,\tau}$ in $O(n)$ time and in $O(\log n)$ bits of working space.*

Proof. We proceed as in the proof of Lemma 31, scanning $T' = T\#$ from left to right. Assume that we are at position i of T' , and assume that we know $\text{DS}[i]$. Then, $aW = T'[i..i + \text{DS}[i] - 2]$ occurs more than τ times in T' and it is a right-maximal substring of T' . To compute $\text{DS}[i+1]$, we take the suffix link from the node of the suffix tree of T' that corresponds to aW , using operation `contractLeft`, and we issue `extendRight` operations on string W using characters $T'[i + \text{DS}[i] - 1]$, $T'[i + \text{DS}[i]]$, etc., until the frequency of the right-extension of W drops again below $\tau + 1$. \square

Lemma 35. *Let $S \in [1..\sigma]^n$ and $T \in [1..\sigma]^m$ be two strings. Given a bidirectional BWT index of their concatenation $S\#_1T\#_2$ that supports every operation in time linear in the size of its output, we can build $\text{MS}_{T,S,\tau}$ in $O(n+m)$ time and in $n+m+o(n+m)$ bits of working space.*

Proof. We use the same algorithm as in Lemma 34, scanning T from left to right and checking at each step the frequency of the current string in S . This can be done in constant time using a bitvector `which`[$1..n+m+2$] indexed to support rank operations, such that `which`[i] = 1 iff the suffix of $S\#_1T\#_2$ with lexicographic rank i starts inside S . \square

By plugging Theorem 12 into Lemmas 34 and 35, we immediately get the following result:

Theorem 14. *Let $S \in [1..\sigma]^n$ and $T \in [1..\sigma]^m$ be two strings. We can build $\text{DS}_{T,\tau}$ in randomized $O(m)$ time and in $O(m \log \sigma)$ bits of working space, and we can build $\text{MS}_{T,S,\tau}$ in randomized $O(n+m)$ time and in $O((n+m) \log \sigma)$ bits of working space.*

Moreover, using Algorithm 3, we can achieve the same bounds in deterministic linear time:

Theorem 15. *Let $S \in [1..\sigma]^n$ and $T \in [1..\sigma]^m$ be two strings. We can build $\text{DS}_{T,\tau}$ in $O(m)$ time and in $O(m \log \sigma)$ bits of working space, and we can build $\text{MS}_{T,S,\tau}$ in $O(n + m)$ time and in $O((n + m) \log \sigma)$ bits of working space.*

Proof. For simplicity we describe just how to compute $\text{MS}_{T,S,1}$. Note that array $\text{MS}_{T,S}$ can be built in linear time from two bitvectors **start** and **end**, of size $|T|$ each, where **start** $[i] = 1$ iff $\text{MS}_{T,S}[i] > \text{MS}_{T,S}[i - 1] - 1$, and where **end** $[j] = 1$ iff there is an i such that $j = i + \text{MS}_{T,S}[i] - 1$.

To build **start**, we use an auxiliary bitvector **start'** of size $|T| + 1$, initialized to zeros, and we run Algorithm 3 to iterate over all right-maximal substrings W of $S\#_1T\#_2$ that occur both in S and in T . Let $\text{repr}'(W) = (\{\text{chars}^S, \text{chars}^T\}, \{\text{first}^S, \text{first}^T\})$. If $\text{chars}^T \setminus \text{chars}^S = \emptyset$, we don't process W further and we continue the iteration. Otherwise, for every character $b \in \text{chars}^T \setminus \text{chars}^S$, we enumerate all the distinct characters a that occur to the left of Wb in T , and their corresponding intervals in $\text{BWT}_{T\#}$, as described in Lemma 21. If aW is not a prefix of a rotation of S , we set to one all positions in **start'** $[i..j]$, where $[i..j]$ is the interval of aWb in $\text{BWT}_{T\#}$. At the end of this process, we invert $\text{BWT}_{T\#}$ and we set **start** $[i + 1] = 1$ for every i such that **start'** $[j] = 1$ and j is the lexicographic rank of suffix $T[i..|T|]\#$ among all suffixes of $T\#$. Finally, we repeat the entire process using $\text{BWT}_{T\#}$, $\text{BWT}_{S\#}$ and **end'**. The claimed complexity comes from Theorems 8 and 6. \square

7.2 Maximal repeats, maximal unique matches, maximal exact matches.

Recall from Section 2 that string W is a *maximal repeat* of string $T \in [1..\sigma]^n$ if W is both left-maximal and right-maximal in T . Let $\{W^1, W^2, \dots, W^{\text{occ}}\}$ be the set of all **occ** distinct maximal repeats of T . We encode such set as a list of **occ** pairs of words $(p^i, |W^i|)$, where p^i is the starting position of an occurrence of W^i in T .

Theorem 16. *Given a string $T \in [1..\sigma]^n$, we can compute an encoding of all its **occ** distinct maximal repeats in $O(n + \text{occ})$ time and in $O(n \log \sigma)$ bits of working space.*

Proof. Recall from Section 4 the representation $\text{repr}(W)$ of a substring W of T . Algorithm 2 invokes function **callback** on every right-maximal substring W of T : inside such function we can determine the left-maximality of W by checking whether $h > 1$, and in the positive case we append pair $(\text{first}[1], |W|)$ to a list **pairs**, where **first** $[1]$ in $\text{repr}(W)$ is the first position of the interval of W in $\text{BWT}_{T\#}$ (see Algorithm 5). After the execution of the whole Algorithm 2, we feed **pairs** to Lemma 3, obtaining in output a list of lengths and starting positions in T that uniquely identifies the set of all maximal repeats of T .

We build $\text{BWT}_{T\#}$ from T using Theorem 8. Then, we use Lemma 7 to build a data structure that supports **access** and **partialRank** queries on $\text{BWT}_{T\#}$, and we discard $\text{BWT}_{T\#}$. We use this structure both to implement function **LF** in Lemma 3, and to build the **rangeDistinct** data structure of Lemma 19. Finally, as described in Theorem 3, we implement Lemma 22 with this **rangeDistinct** data structure. We allocate the space for **pairs** and for related data structures in Lemma 3 using the static allocation strategy described in Section 3.1. We charge to the output the space taken by **pairs**. In Lemma 3, we charge to the output the space taken by list **translate**, as well as part of the working space used by radix sort. \square

Maximal repeats have been detected from the input string in $O(n \log \sigma)$ bits of working space before, but not in overall $O(n)$ time. Specifically, it is possible to achieve overall running time $O(n \log \sigma)$ by combining the BWT construction algorithm described in [38], which runs in $O(n \log \log \sigma)$ time, with the maximal repeat detection algorithm described in [12], which runs in $O(n \log \sigma)$ time. The claim of Theorem 16 holds also for an encoding of the maximal repeats that contains, for every maximal repeat, the starting position of *all* its occurrences in T . In this case, **occ** becomes the number of *occurrences* of all maximal repeats of T . Specifically, given the BWT interval of a maximal repeat W , it suffices to mark all the positions inside the interval in a bitvector **marked** $[1..n]$, to assign a unique identifier to every distinct maximal repeat, and to sort the translated list **pairs** by such identifier before returning it in output. Bitvector **marked** can be replaced by a smaller bitvector **marked'** as described in Lemma 3.

Once we have the encoding $(p^i, |W^i|)$ of every maximal repeat W^i , we can return the corresponding *string* W^i by scanning T in blocks of size $\log n$, i.e. outputting $\log_\sigma n$ characters in constant time: this allows us to

ALGORITHM 5: Function `callback` for maximal repeats. See Theorem 16 and Algorithm 2.

Input: `repr(W)`, $|W|$, BWT_T , and C array of string $T \in [1..\sigma]^{n-1}\#$. Matrices A , F , L , `gamma`, `leftExtensions`, and counter h , from Lemma 21. List `pairs`.

```

1 if  $h < 2$  then
2   | return;
3 end
4 pairs.append((first[1], |W|));

```

print the C total characters in the output in overall $C/\log_\sigma n$ time. Alternatively, we can discard the original string altogether, and maintain instead an auxiliary stack of characters while we traverse the suffix-link tree in Lemma 22. Once we detect a maximal repeat, we print its string to the output by scanning the auxiliary stack in blocks of size $\log n$. Recall from Section 2.3 that the leaves of the suffix-link tree are maximal repeats: this implies that the depth d of the auxiliary stack is at most equal to the length of the longest maximal repeat, thus the maximum size $d \log \sigma$ of the auxiliary stack can be charged to the output.

A *supermaximal repeat* is a maximal repeat that is not a substring of another maximal repeat, and a *near-supermaximal repeat* is a maximal repeat that has at least one occurrence that is not contained inside an occurrence of another maximal repeat (see e.g. [32]). The proof of Theorem 16 can be adapted to detect supermaximal and near-supermaximal repeats within the same bounds: we leave the details to the reader.

Consider two string S and T . For a *maximal unique match (MUM)* W between $S \in [1..\sigma]^n$ and $T \in [1..\sigma]^m$, it holds that: (1) $W = S[i..i+k-1]$ and $W = T[j..j+k-1]$ for exactly one $i \in [1..n]$ and for exactly one $j \in [1..m]$; (2) if $i-1 \geq 1$ and $j-1 \geq 1$, then $S[i-1] \neq T[j-1]$; (4) if $i+k \leq n$ and $j+k \leq m$, then $S[i+k] \neq T[j+k]$ (see e.g. [32]). To detect all the MUMs of S and T , it would suffice to build the suffix tree of the concatenation $C = S\#_1T\#_2$ and to traverse its internal nodes, since MUMs are right-maximal substrings of C , like maximal repeats. More specifically, only internal nodes v with exactly two leaves as children can be MUMs. Let the two leaves of a node v be associated with suffixes $C[i..|C|]$ and $C[j..|C|]$, respectively. Then, i and j must be such that $i \leq |S|$ and $j > |S|+1$, and the left-maximality of v can be checked by accessing $S[i-1 \pmod n]$ and $T[j-1 \pmod m]$ in constant time.

This notion extends naturally to a set of strings: a string W is a maximal unique match (MUM) of d strings T^1, T^2, \dots, T^d , where $T^i \in [1..\sigma]^{n_i}$, if W occurs exactly once in T^i for all $i \in [1..d]$, and if W cannot be extended to the left or to the right without losing one of its occurrences. We encode the set of all maximal unique matches W of T^1, T^2, \dots, T^d as a list of `occ` triplets of words $(p^i, |W|, \text{id})$, where p^i is the first position of the occurrence of W in string T^i , and `id` is a number that uniquely identifies W . Note that the maximal unique matches of T^1, T^2, \dots, T^d are maximal repeats of the concatenation $T = T^1\#_1T^2\#_1 \dots \#_1T^d\#_2$, thus we can adapt Theorem 16 as follows:

Theorem 17. *Given a set of strings T^1, T^2, \dots, T^d where $d > 1$ and $T^i \in [1..\sigma]^+$ for all $i \in [1..d]$, we can compute an encoding of all the distinct maximal unique matches of the set in $O(n + \text{occ})$ time and in $O(n \log \sigma)$ bits of working space, where $n = \sum_{i=1}^d |T^i|$ and `occ` is the number of words in the encoding.*

Proof. We build the same data structures as in Theorem 16, but on string $T = T^1\#_1T^2\#_1 \dots \#_1T^d\#_2$, and we enumerate all the maximal repeats of T using Algorithm 2. Whenever we find a maximal repeat W with exactly d occurrences in T , we set to one in a bitvector `intervals`[1..|T|] the first and the last position of the interval of W in BWT_T (see Algorithm 6). Note that the BWT intervals of all the maximal repeats of T with exactly d occurrences are disjoint. Then, we index `intervals` to support rank queries in constant time, we allocate another bitvector `documents`[1..|T|], and we invert BWT_T . Assume that, at the generic step of the inversion, we are at position i in T and at position j in BWT_T . We decide whether j belongs to the interval of a maximal repeat with d occurrences, by checking whether `rank1(intervals, j)` is odd, or, if it is even, whether `intervals[j] = 1`. If j belongs to an interval $[x..y]$ that has been marked in `intervals`, we compute x using rank queries on `intervals`, and we set `documents[x + p - 1]` to one, where p is the identifier of the document that contains position i in T . Finally, we scan bitvectors `intervals` and `documents` synchronously: for each interval $[x..y]$ that has been marked in `intervals` and such that

ALGORITHM 6: First callback function for maximal unique matches. See Theorem 17 and Algorithm 2.

Input: $\text{repr}(W)$, $|W|$, BWT_T , and C array of string $T \in [1..\sigma]^{n-1}\#$. Matrices A , F , L , gamma , leftExtensions , and counter h , from Lemma 21. Bitvector $\text{intervals}[1..|T|]$.

```

1 if  $h < 2$  or  $\text{first}[|\text{first}|] - \text{first}[1] \neq d$  then
2   | return;
3 end
4  $\text{intervals}[\text{first}[1]] \leftarrow 1$ ;
5  $\text{intervals}[\text{first}[|\text{first}|] - 1] \leftarrow 1$ ;

```

ALGORITHM 7: Second callback function for maximal unique matches. See Theorem 17 and Algorithm 2.

Input: $\text{repr}(W)$, $|W|$, BWT_T , and C array of string $T \in [1..\sigma]^{n-1}\#$. Matrices A , F , L , gamma , leftExtensions , and counter h , from Lemma 21. Bitvector $\text{intervals}[1..|T|]$. List pairs . Integer id .

```

1 if  $h < 2$  or  $\text{first}[|\text{first}|] - \text{first}[1] \neq d$  or  $\text{intervals}[\text{first}[1]] \neq 1$  or  $\text{intervals}[\text{first}[|\text{first}|] - 1] \neq 1$  then
2   | return;
3 end
4 for  $i \in [\text{first}[1]..\text{first}[|\text{first}|] - 1]$  do
5   |  $\text{pairs.append}((i, |W|, \text{id}))$ ;
6   |  $\text{id} \leftarrow \text{id} + 1$ ;
7 end

```

$\text{documents}[i] = 0$ for some $i \in [x..y]$, we reset $\text{intervals}[x]$ and $\text{intervals}[y]$ to zero. Finally, we iterate again over all the maximal repeats of T with exactly d occurrences, using Algorithm 2. Let W be such a maximal repeat, with interval $[x..y]$ in BWT_T : if $\text{intervals}[x] = \text{intervals}[y] = 1$, we append to list pairs of Theorem 16 a triplet $(i, |W|, \text{id})$ for all $i \in [x..y]$, where id is a number that uniquely identifies W (see Algorithm 7). Then, we continue as in Theorem 16. \square

Maximal exact matches (MEMs) are related to maximal repeats as well. A triplet (i, j, ℓ) is a *maximal exact match* (also called *maximal pair*) of two strings T^1 and T^2 if: (1) $T^1[i \dots i + \ell - 1] = T^2[j \dots j + \ell - 1] = W$; (2) if $i - 1 \geq 1$ and $j - 1 \geq 1$, then $T^1[i - 1] \neq T^2[j - 1]$; (3) if $i + \ell \leq |T^1|$ and $j + \ell \leq |T^2|$, then $T^1[i + \ell] \neq T^2[j + \ell]$ (see e.g. [3, 32]). We encode the set of all maximal exact matches of strings T^1 and T^2 as a list of occ such triplets. Since W is a maximal repeat of $T^1\#_1T^2\#_2$ that occurs both in T^1 and in T^2 , we can build a detection algorithm on top of the generalized iterator of Algorithm 3, as follows:

Theorem 18. *Given two strings T^1 and T^2 in $[1..\sigma]^+$, we can compute an encoding of all their occ maximal exact matches in $O(|T^1| + |T^2| + \text{occ})$ time and in $O((|T^1| + |T^2|)\log \sigma)$ bits of working space.*

Proof. Recall that Algorithm 3 uses a `rangeDistinct` data structure built on top of the BWT of T^1 , and a `rangeDistinct` data structure built on top of the BWT of T^2 , to iterate over all the right-maximal substrings W of $T^1\#_1T^2\#_2$. For every such W , the algorithm gives to function `callback` the intervals of all strings aWb such that $a \in [1..\sigma]$, $b \in [1..\sigma]$, and aWb is a prefix of a rotation of T^1 , and with the intervals of all strings cWd such that $c \in [1..\sigma]$, $d \in [1..\sigma]$, and cWd is a prefix of a rotation of T^2 . Recall from Section 4 the representation $\text{repr}'(W)$ of a substring W of $T^1\#_1T^2\#_2$. Inside function `callback`, it suffices to determine whether W occurs in both T^1 and T^2 , using arrays first^1 and first^2 of $\text{repr}'(W)$, and to determine whether W is left-maximal in $T^1\#_1T^2\#_2$, by checking whether $h > 1$ (see Algorithm 8). If both such tests succeed, we build a set X that represents all strings aWb that are the prefix of a rotation of T^1 , and a set X^2 that represents all strings cWd that are the prefix of a rotation of T^2 :

$$\begin{aligned}
X^1 &= \{ (a, b, i, j) : a = \text{leftExtensions}[p], p \leq h, \text{gamma}^1[a] > 0, b = A^1[a][q], \\
&\quad q \leq \text{gamma}^1[a], i = F^1[a][q], j = L^1[a][q] \} \\
X^2 &= \{ (c, d, i', j') : c = \text{leftExtensions}[p], p \leq h, \text{gamma}^2[c] > 0, d = A^2[c][q], \\
&\quad q \leq \text{gamma}^1[c], i' = F^2[c][q], j' = L^2[c][q] \}
\end{aligned}$$

ALGORITHM 8: Function `callback` for maximal exact matches. See Theorem 18 and Algorithm 3. Operator \otimes is from Lemma 36.

Input: `repr'(W)`, $|W|$, $\{\text{BWT}_{T^i\#}\}$, $\{C^i\}$ arrays. Matrices $\{A^i\}$, $\{F^i\}$, $\{L^i\}$, $\{\text{gamma}^i\}$. Array `leftExtensions` and counter h . List `pairs`.

```

1 if  $h < 2$  or  $|\text{chars}^1| = 0$  or  $|\text{chars}^2| = 0$  then
2   return;
3 end
4  $X^1 \leftarrow \emptyset$ ;
5  $X^2 \leftarrow \emptyset$ ;
6 for  $i \in [1..h]$  do
7    $a \leftarrow \text{leftExtensions}[i]$ ;
8   if  $\text{gamma}^1[a] > 0$  then
9     for  $j \in [1..\text{gamma}^1[a]]$  do
10       $X^1 \leftarrow X^1 \cup \{(a, A^1[a][j], F^1[a][j], L^1[a][j])\}$ ;
11    end
12  end
13  if  $\text{gamma}^2[a] > 0$  then
14    for  $j \in [1..\text{gamma}^2[a]]$  do
15       $X^2 \leftarrow X^2 \cup \{(a, A^2[a][j], F^2[a][j], L^2[a][j])\}$ ;
16    end
17  end
18 end
19  $Y \leftarrow X^1 \otimes X^2$ ;
20 for  $(i, j, i', j') \in Y$  do
21   for  $x \in [i..j]$ ,  $y \in [i'..j']$  do
22     pairs.append((x, y, |W|));
23   end
24 end

```

In such sets, $[i..j]$ is the interval of aWb in the BWT of $T^1\#$, and $[i'..j']$ is the interval of cWd in the BWT of $T^2\#$. Building X^1 and X^2 for all maximal repeats W of $T^1\#_1T^2\#_2$ takes overall linear time in the size of the input, since every element of X^1 (respectively, of X^2) can be charged to a distinct edge or implicit Weiner link of the generalized suffix tree of $T^1\#_1T^2\#_2$, and the number of such objects is linear in the size of the input (see Observation 1). Then, we use Lemma 36 to compute the set of all quadruplets (i, j, i', j') such that $(a, b, i, j) \in X^1$, $(c, d, i', j') \in X^2$, $a \neq c$ and $b \neq d$, in overall linear time in the size of the input and of the output, and for every such quadruplet we append all triplets $(x, y, |W|)$ to list `pairs` of Theorem 16, where $x \in [i..j]$ and $y \in [i'..j']$. Running Algorithm 3 and building its input data structures from T^1 and T^2 takes overall $O(|T^1| + |T^2|)$ time and $O((|T^1| + |T^2|) \log \sigma)$ bits of working space, by combining Theorem 8 with Lemmas 7, 19 and 23.

Finally, we translate every x and y in `pairs` to a string position, as described in Theorem 16. We allocate the space for `pairs` and for related data structures in Lemma 3 using the static allocation strategy described in Section 3.1. We charge to the output the space taken by `pairs`. In Lemma 3, we charge to the output the space taken by list `translate`, as well as part of the working space used by radix sort. \square

Lemma 36. *Let Σ be a set, and let A and B be two subsets of $\Sigma \times \Sigma$. We can compute $A \otimes B = \{(a, b, c, d) \mid (a, b) \in A, (c, d) \in B, a \neq c, b \neq d\}$ in $O(|A| + |B| + |A \otimes B|)$ time.*

Proof. We assume without loss of generality that $|A| < |B|$. We say that two pairs (a, b) , (c, d) are *compatible* if $a \neq c$ and $b \neq d$. Note that, if (a, b) and (c, d) are compatible, then the only elements of $\Sigma \times \Sigma$ that are incompatible with both (a, b) and (c, d) are (a, d) and (c, b) . We iteratively select a pair $(a, b) \in A$ and scan A in $O(|A|) = O(|B|)$ time to find another compatible pair (c, d) : if we find one, we scan B and report every pair in B that is compatible with either (a, b) or (c, d) . The output will be of size $|B| - 2$ or larger,

thus the time to scan A and B can be charged to the output. Then, we remove (a, b) and (c, d) from A and repeat the process. If A becomes empty we stop. If all the remaining pairs in A are incompatible with our selected pair (a, b) , that is, if $c = a$ or $d = b$ for every $(c, d) \in A$, we build subsets A^a and A^b where $A^a = \{(a, x) : x \neq b\} \subseteq A$ and $A^b = \{(x, b) : x \neq a\} \subseteq A$. Then we scan B , and for every pair $(x, y) \in B$ different from (a, b) we do the following. If $x \neq a$ and $y \neq b$, then we report (a, b, x, y) , $\{(a, z, x, y) : (a, z) \in A^a, z \neq y\}$ and $\{(z, b, x, y) : (z, b) \in A^b, z \neq x\}$. Pairs $(a, y) \in A^a$ and $(x, b) \in A^b$ are the only ones that do not produce output, thus the cost of scanning A^a and A^b can be charged to printing the result. If $x = a$ and $y \neq b$, then we report $\{(z, b, x, y) : (z, b) \in A^b\}$. If $x \neq a$ and $y = b$, then we report $\{(a, z, x, y) : (a, z) \in A^a\}$. \square

Theorem 18 uses the matrices and arrays of Lemma 21 to access all the left-extensions aW of a string W , and for every such left-extension to access all its right-extensions aWb . A similar approach can be used to compute all the *minimal absent words* of a string T . String W is a *minimal absent word* of a string $T \in \Sigma^+$ if W is not a substring of T and if every proper substring of W is a substring of T (see e.g. [17]). To decide whether aWb is a minimal absent word of T , where $\{a, b\} \subseteq \Sigma$, it suffices to check that aWb does not occur in T , and that both aW and Wb occur in T . Only a maximal repeat of T can be the infix W of a minimal absent word aWb : we can enumerate all the maximal repeats W of T as in Theorem 16. Recall also that aWb is a minimal absent word of T only if both aW and Wb occur in T . We can use `repr(W)` to enumerate all strings Wb that occur in T , we can use vector `leftExtensions` to enumerate all strings aW that occur in T , and finally we can use matrix A to discard all strings aWb that occur in T . Algorithm 9 uses this approach to output an encoding of all distinct minimal absent words of T as a list of triplets (i, ℓ, b) , where each triplet encodes minimal absent word $T[i..i + \ell - 1] \cdot b$. Every operation of this algorithm can be charged to an element of the output, to an edge of the suffix tree of T , or to a Weiner link. The following theorem holds by this observation, and by applying the same steps as in Theorem 16: we leave its proof to the reader.

Theorem 19. *Given a string $T \in [1..\sigma]^n$, we can compute an encoding of all its `occ` minimal absent words in $O(n + \text{occ})$ time and in $O(n \log \sigma)$ bits of working space.*

Recall from Section 2 that `occ` can be of size $\Theta(n\sigma)$ in this case. Minimal absent words have been detected in linear time in the length of the input before, but using a suffix array (see [4] and references therein).

7.3 String kernels

Another way of comparing and analyzing strings consists in studying the composition and abundance of all the distinct strings that occur in them. Given two strings T^1 and T^2 , a *string kernel* is a function that simultaneously converts T^1 and T^2 to *composition vectors* $\{\mathbf{T}^1, \mathbf{T}^2\} \subset \mathbb{R}^n$, indexed by a given set of $n > 0$ distinct strings, and that computes a similarity or a distance measure between \mathbf{T}^1 and \mathbf{T}^2 (see e.g. [35, 45]). Value $\mathbf{T}^1[W]$ is typically a function of the number $f_{T^1}(W)$ of (possibly overlapping) occurrences of string W in T^1 (for example the estimate $p_i(W) = f_{T^1}(W)/(|T^1| - |W| + 1)$ of the empirical probability of observing W in T^1). In this section, we focus on computing the cosine of the angle between \mathbf{T}^1 and \mathbf{T}^2 , defined as:

$$\kappa(\mathbf{T}^1, \mathbf{T}^2) = \frac{\sum_W \mathbf{T}^1[W] \mathbf{T}^2[W]}{\sqrt{(\sum_W \mathbf{T}^1[W]^2)(\sum_W \mathbf{T}^2[W]^2)}}$$

Specifically, we consider the case in which \mathbf{T}^i is indexed by all distinct strings of a given length k (called *k-mers*), and the case in which \mathbf{T}^i is indexed by all distinct strings of *any length*:

Definition 11. *Given a string $T \in [1..\sigma]^+$ and a length $k > 0$, let vector $\mathbf{T}_k = [1..\sigma^k]$ be such that $\mathbf{T}_k[W] = f_T(W)$ for every $W \in [1..\sigma]^k$. The *k-mer complexity* $C(T, k)$ of string T is the number of nonzero components of \mathbf{T}_k . The *k-mer kernel* between two strings T^1 and T^2 is $\kappa(\mathbf{T}_k^1, \mathbf{T}_k^2)$.*

Definition 12. *Given a string $T \in [1..\sigma]^+$, consider the infinite-dimensional vector \mathbf{T}_∞ , indexed by all distinct substrings $W \in [1..\sigma]^+$, such that $\mathbf{T}_\infty[W] = f_T(W)$. The *substring complexity* $C(T)$ of string T is the number of nonzero components of \mathbf{T}_∞ . The *substring kernel* between two strings T^1 and T^2 is $\kappa(\mathbf{T}_\infty^1, \mathbf{T}_\infty^2)$.*

ALGORITHM 9: Function `callback` for minimal absent words. See Theorem 19 and Algorithm 2.

Input: `repr(W)`, $|W|$, BWT_T , and C array of string $T \in [1..\sigma]^{n-1}\#$. Matrices A , F , L , `gamma`, `leftExtensions`, and counter h , from Lemma 21. Bitvector `used` $[1..\sigma]$ initialized to all zeros. List `pairs`.

```
1 if  $h < 2$  then
2   | return;
3 end
4 for  $i \in [1..|\text{chars}|]$  do
5   | used $[\text{chars}[i]] \leftarrow 1$ ;
6 end
7 for  $i \in [1..h]$  do
8   |  $a \leftarrow \text{leftExtensions}[i]$ ;
9   | for  $j \in [1..\text{gamma}[a]]$  do
10    | | used $[A[a][j]] \leftarrow 0$ ;
11    | end
12    | for  $j \in [1..|\text{chars}|]$  do
13    | |  $b \leftarrow \text{chars}[j]$ ;
14    | | if used $[b] = 0$  then
15    | | | pairs.append $((F[a][1], |W| + 1, b))$ ;
16    | | | used $[b] \leftarrow 1$ ;
17    | | end
18    | end
19 end
20 for  $i \in [1..|\text{chars}|]$  do
21   | used $[\text{chars}[i]] \leftarrow 0$ ;
22 end
```

Substring complexity and substring kernels, with or without a constraint on string length, can be computed using the suffix tree of a single string or the generalized suffix tree of two strings, using a *telescoping technique* that works by adding and subtracting terms to and from a sum, and that does not depend on the order in which the nodes of the suffix tree are enumerated [9]. We can thus implement all such algorithms as callback functions of Algorithms 2 and 3:

Theorem 20. *Given a string $T \in [1..\sigma]^n$, there is an algorithm that computes:*

- *the k -mer complexity $C(T, k)$ of T , in $O(n)$ time and in $O(n \log \sigma)$ bits of working space, for a given integer k ;*
- *the substring complexity $C(T)$, in $O(n)$ time and in $O(n \log \sigma)$ bits of working space.*

Given two strings T^1 and T^2 in $[1..\sigma]^+$, there is an algorithm that computes:

- *the k -mer kernel between T^1 and T^2 , in $O(|T^1| + |T^2|)$ time and $O((|T^1| + |T^2|) \log \sigma)$ bits of working space, for a given integer k ;*
- *the substring kernel between T^1 and T^2 , in $O(|T^1| + |T^2|)$ time and in $O((|T^1| + |T^2|) \log \sigma)$ bits of working space.*

Proof. To make the paper self-contained, we just sketch the proof of k -mer complexity given in [9]: the same telescoping technique can be applied to solve all other problems: see [9].

A k -mer of T is either the label of a node of the suffix tree of T , or it ends in the middle of an edge (u, v) of the suffix tree. In the latter case, we assume that the k -mer is represented by its locus v , which might be a leaf. Let $C(T, k)$ be initialized to $|T| + 1 - k$, i.e. to the number of leaves that correspond to suffixes of $T\#$ of length at least k , excluding suffix $T[|T| - k + 2..|T|]\#$. We use Algorithm 2 to enumerate the internal nodes of $\text{ST}_{T\#}$, and every time we enumerate a node v we proceed as follows. Let $\ell(v) = W$. If $|W| < k$ we

leave $C(T, k)$ unaltered, otherwise we increment $C(T, k)$ by one and we decrement $C(T, k)$ by the number of children of v in $\mathbf{ST}_{T\#}$, which is equal to $|\mathbf{chars}|$ in $\mathbf{repr}(W)$. It follows that every node v of $\mathbf{ST}_{T\#}$ that is located at depth at least k and that is not the locus of a k -mer is both added to $C(T, k)$ (when the algorithm visits v) and subtracted from $C(T, k)$ (when the algorithm visits $\mathbf{parent}(v)$). Leaves at depth at least k are added by the initialization of $C(T, k)$, and subtracted during the enumeration. Conversely, every locus v of a k -mer of T (including leaves) is just added to $C(T, k)$, because $|\ell(\mathbf{parent}(v))| < k$. The claimed complexity comes from Theorem 8 and Theorem 3. \square

A number of other kernels and complexity measures can be implemented on top of Algorithms 2 and 3: see [9] for details. Since such iteration algorithms work on data structures that can be built from the input strings in deterministic linear time, all such kernels and complexity measures can be computed from the input strings in deterministic $O(n)$ time and in $O(n \log \sigma)$ bits of working space, where n is the total length of the input strings.

Acknowledgement

The authors wish to thank Travis Gagie for explaining the data structure built in Lemma 33, as well as for valuable comments and encouragements, Gonzalo Navarro for explaining the algorithm in Theorem 17, Enno Ohlebusch for useful comments and remarks, and Alexandru Tomescu for valuable comments and encouragements.

References

- [1] Amihood Amir, Gad M Landau, Moshe Lewenstein, and Dina Sokol. Dynamic text and static pattern matching. *ACM Transactions on Algorithms (TALG)*, 3(2), 2007.
- [2] Alberto Apostolico. The Myriad Virtues of Subword Trees. In A. Apostolico and Z. Galil, editors, *Combinatorial Algorithms on Words*, NATO Advance Science Institute Series F: Computer and Systems Sciences, pages 85–96, Berlin, Heidelberg, 1985. Springer-Verlag.
- [3] Brenda S Baker. On finding duplication and near-duplication in large software systems. In *Reverse Engineering, 1995., Proceedings of 2nd Working Conference on*, pages 86–95. IEEE, 1995.
- [4] Carl Barton, Alice Heliou, Laurent Mouchard, and Solon P Pissis. Linear-time computation of minimal absent words using suffix array. *arXiv preprint arXiv:1406.6341*, 2014.
- [5] D. Belazzougui and G. Navarro. Alphabet-independent compressed text indexing. In *Proc. European Symposium on Algorithms (ESA 2011)*, pages 748–759. ACM, 2011.
- [6] Djamal Belazzougui, Paolo Boldi, Rasmus Pagh, and Sebastiano Vigna. Monotone minimal perfect hashing: searching a sorted table with $o(1)$ accesses. In *Proc. ACM-SIAM Symposium on Discrete Algorithms (SODA 2009)*, pages 785–794, USA, 2009. ACM-SIAM.
- [7] Djamal Belazzougui, Paolo Boldi, Rasmus Pagh, and Sebastiano Vigna. Theory and practice of monotone minimal perfect hashing. *Journal of Experimental Algorithmics (JEA)*, 16:3–2, 2011.
- [8] Djamal Belazzougui and Fabio Cunial. Indexed matching statistics and shortest unique substrings. In *Proc. Symposium on String Processing and Information Retrieval (SPIRE 2014)*, pages 179–190, Brazil, 2014. Springer.
- [9] Djamal Belazzougui and Fabio Cunial. A framework for space-efficient string kernels. In *Annual Symposium on Combinatorial Pattern Matching*, pages 13–25. Springer, 2015.
- [10] Djamal Belazzougui and Gonzalo Navarro. Alphabet-independent compressed text indexing. *ACM Transactions on Algorithms*, 10(4):23:1–23:19, 2014.
- [11] Djamal Belazzougui, Gonzalo Navarro, and Daniel Valenzuela. Improved compressed indexes for full-text document retrieval. *Journal of Discrete Algorithms*, 18:3–13, 2013.
- [12] Timo Beller, Katharina Berger, and Enno Ohlebusch. Space-efficient computation of maximal and supermaximal repeats in genome sequences. In *19th International Symposium on String Processing and Information Retrieval (SPIRE 2012)*, volume 7608 of *Lecture Notes in Computer Science*, pages 99–110. Springer, 2012.
- [13] Timo Beller, Simon Gog, Enno Ohlebusch, and Thomas Schnattinger. Computing the longest common prefix array based on the burrows-wheeler transform. *J. Discrete Algorithms*, 18:22–31, 2013.
- [14] Andrej Brodnik. Computation of the least significant set bit. In *Proc. 2nd Electrotechnical and Computer Science Conference*, volume 90, Portoroz, Slovenia, 1993.
- [15] M. Burrows and D. Wheeler. A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, 1994.
- [16] D. Clark. *Compact Pat Trees*. PhD thesis, University of Waterloo, Canada, 1996.
- [17] Maxime Crochemore, Filippo Mignosi, and Antonio Restivo. Automata and forbidden words. *Information Processing Letters*, 67(3):111–117, 1998.
- [18] Peter Elias. Efficient storage and retrieval by content and address of static files. *J. ACM*, 21(2):246–260, 1974.
- [19] Peter Elias. Universal codeword sets and representations of the integers. *Information Theory, IEEE Transactions on*, 21(2):194–203, 1975.
- [20] Robert M. Fano. On the number of bits required to implement an associative memory. Memorandum 61, Computer Structures Group, Project MAC, MIT, Cambridge, Mass., n.d., 1971.
- [21] Martin Farach. Optimal suffix tree construction with large alphabets. In *Proc. Symposium on Foundations of Computer Science (FOCS 1997)*, pages 137–143, Miami Beach, Florida, USA, 1997. IEEE Computer Society.
- [22] P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *Proc. 41st IEEE Symposium on Foundations of Computer Science (FOCS 2000)*, pages 390–398, USA, 2000. IEEE.
- [23] P. Ferragina and G. Manzini. Indexing compressed texts. *Journal of the ACM*, 52(4):552–581, 2005.

- [24] Paolo Ferragina and Rossano Venturini. A simple storage scheme for strings achieving entropy bounds. *Theoretical Computer Science*, 372:115–121, 2007.
- [25] Johannes Fischer. Optimal succinctness for range minimum queries. In *Proc. Latin American Theoretical Informatics Symposium (LATIN 2010)*, pages 158–169. Springer, Mexico, 2010.
- [26] Johannes Fischer. Combined data structure for previous- and next-smaller-values. *Theor. Comput. Sci.*, 412(22):2451–2456, 2011.
- [27] Alexander Golynski, J Ian Munro, and S Srinivasa Rao. Rank/select operations on large alphabets: a tool for text indexing. In *Proc. ACM-SIAM Symposium on Discrete Algorithms (SODA 2006)*, pages 368–373, USA, 2006. ACM.
- [28] R. Grossi and J. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. In *Proc. 32nd ACM Symposium on Theory of Computing (STOC)*, pages 397–406, 2000.
- [29] Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. High-order entropy-compressed text indexes. In *Proc. ACM-SIAM Symposium on Discrete Algorithms (SODA 2003)*, pages 841–850, USA, 2003. Society for Industrial and Applied Mathematics.
- [30] Roberto Grossi, Alessio Orlandi, Rajeev Raman, and S Srinivasa Rao. More haste, less waste: Lowering the redundancy in fully indexable dictionaries, 2009.
- [31] Roberto Grossi and Jeffrey Scott Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM Journal on Computing*, 35(2):378–407, 2005.
- [32] D. Gusfield. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*. Cambridge University Press, Cambridge, UK, 1997.
- [33] T. Hagerup and T. Tholey. Efficient minimal perfect hashing in nearly minimal space. In *Proc. Symposium on Theoretical Aspects of Computer Science (STACS 2001)*, pages 317–326, Dresden, Germany, 2001. Springer-Verlag.
- [34] Torben Hagerup, Peter Bro Miltersen, and Rasmus Pagh. Deterministic dictionaries. *Journal of Algorithms*, 41(1):69–85, 2001.
- [35] David Haussler. Convolution kernels on discrete structures. Technical report, Technical report, UC Santa Cruz, 1999.
- [36] Charles AR Hoare. Quicksort. *The Computer Journal*, 5(1):10–16, 1962.
- [37] Wing-Kai Hon and Kunihiko Sadakane. Space-economical algorithms for finding maximal unique matches. In *Proc. Annual Symp. on Combinatorial Pattern Matching (CPM)*, volume 2373 of *LNCS*, pages 144–152. Springer, 2002.
- [38] Wing-Kai Hon, Kunihiko Sadakane, and Wing-Kin Sung. Breaking a time-and-space barrier in constructing full-text indices. *SIAM Journal on Computing*, 38(6):2162–2178, 2009.
- [39] Juha Kärkkäinen, Peter Sanders, and Stefan Burkhardt. Linear work suffix array construction. *Journal of the ACM (JACM)*, 53(6):918–936, 2006.
- [40] Dong Kyue Kim, Jeong Seop Sim, Heejin Park, and Kunsoo Park. Constructing suffix arrays in linear time. *Journal of Discrete Algorithms*, 3(2):126–142, 2005.
- [41] Pang Ko and Srinivas Aluru. Space efficient linear time construction of suffix arrays. In *Proc. Symposium on Combinatorial Pattern Matching (CPM 2003)*, pages 200–210, Morelia, Mexico, 2003. Springer.
- [42] M Oguzhan Kulekci, Jeffrey Scott Vitter, and Bojian Xu. Efficient maximal repeat finding using the burrows-wheeler transform and wavelet tree. *IEEE/ACM Transactions on Computational Biology and Bioinformatics (TCBB)*, 9(2):421–429, 2012.
- [43] Tak Wah Lam, Ruiqiang Li, Alan Tam, Simon Wong, Edward Wu, and SM Yiu. High throughput short read alignment via bi-directional BWT. In *BIBM 2009*, pages 31–36, 2009.
- [44] Ruiqiang Li, Chang Yu, Yingrui Li, Tak Wah Lam, Siu-Ming Yiu, Karsten Kristiansen, and Jun Wang. Soap2: An improved ultrafast tool for short read alignment. *Bioinformatics*, 25(15):1966–1967, 2009.
- [45] Huma Lodhi, Craig Saunders, John Shawe-Taylor, Nello Cristianini, and Chris Watkins. Text classification using string kernels. *The Journal of Machine Learning Research*, 2:419–444, 2002.

- [46] Udi Manber and Eugene W. Myers. Suffix arrays: A new method for on-line string searches. *SIAM J. Comput.*, 22(5):935–948, 1993.
- [47] I. Munro. Tables. In *Proc. 16th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, LNCS v. 1180, pages 37–42, 1996.
- [48] J Ian Munro, Rajeev Raman, Venkatesh Raman, and Satti Srinivasa Rao. Succinct representations of permutations. In *Proc. International Colloquium on Automata, Languages and Programming (ICALP 2003)*, pages 345–356. Springer, Eindhoven, The Netherlands, 2003.
- [49] J Ian Munro and Venkatesh Raman. Succinct representation of balanced parentheses and static trees. *SIAM Journal on Computing*, 31(3):762–776, 2001.
- [50] S. Muthukrishnan. Efficient algorithms for document retrieval problems. In *Proc. ACM-SIAM Symposium on Discrete Algorithms (SODA 2002)*, pages 657–666, San Francisco, USA, 2002. ACM-SIAM.
- [51] G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39(1):Article 2, 2007.
- [52] Gonzalo Navarro and Kunihiko Sadakane. Fully functional static and dynamic succinct trees. *ACM Transactions on Algorithms*, 10(3):16:1–16:39, 2014.
- [53] Enno Ohlebusch, Timo Beller, and Mohamed Ibrahim Abouelhoda. Computing the burrows-wheeler transform of a string and its reverse in parallel. *J. Discrete Algorithms*, 25:21–33, 2014.
- [54] Daisuke Okanohara and Kunihiko Sadakane. Practical entropy-compressed rank/select dictionary. In *Proc. Workshop on Algorithm Engineering and Experiments (ALENEX 2007)*, pages 60–70, New Orleans, USA, 2007. SIAM.
- [55] Daisuke Okanohara and Kunihiko Sadakane. A linear-time burrows-wheeler transform using induced sorting. In *Proc. Symposium on String Processing and Information Retrieval (SPIRE 2009)*, volume 5721 of LNCS, pages 90–101, Saarisekä, Finland, 2009. Springer.
- [56] Daisuke Okanohara and Jun’ichi Tsujii. Text categorization with all substring features. In *Proceedings of the 2009 SIAM International Conference on Data Mining (SDM)*, pages 838–846. SIAM, 2009.
- [57] Rasmus Pagh. Low redundancy in static dictionaries with constant query time. *SIAM J. Comput.*, 31(2):353–363, 2001.
- [58] Mihai Patrascu. Succincter. In *Foundations of Computer Science, 2008. FOCS’08. IEEE 49th Annual IEEE Symposium on*, pages 305–313. IEEE, 2008.
- [59] M.M. Robertson. A generalization of quasi-monotone sequences. *Proceedings of the Edinburgh Mathematical Society (Series 2)*, 16(01):37–41, 1968.
- [60] K. Sadakane. Compressed text databases with efficient query algorithms based on the compressed suffix array. In *Proc. 11th International Symposium on Algorithms and Computation (ISAAC)*, LNCS v. 1969, pages 410–421, 2000.
- [61] K. Sadakane and G. Navarro. Fully-functional succinct trees. In *Proc. ACM-SIAM Symposium on Discrete Algorithms (SODA 2010)*, pages 134–149, Austin, Texas, USA, 2010. ACM-SIAM.
- [62] Kunihiko Sadakane. Succinct representations of lcp information and improvements in the compressed suffix arrays. In *Proc. ACM-SIAM Symposium on Discrete Algorithms (SODA 2002)*, pages 225–232, San Francisco, USA, 2002. ACM-SIAM.
- [63] Kunihiko Sadakane. Compressed suffix trees with full functionality. *Theory Comput. Syst.*, 41(4):589–607, 2007.
- [64] Kunihiko Sadakane. Succinct data structures for flexible text retrieval systems. *J. Discrete Algorithms*, 5(1):12–22, 2007.
- [65] Thomas Schnattinger, Enno Ohlebusch, and Simon Gog. Bidirectional search in a string with wavelet trees. In *CPM 2010*, pages 40–50, 2010.
- [66] Thomas Schnattinger, Enno Ohlebusch, and Simon Gog. Bidirectional search in a string with wavelet trees and bidirectional matching statistics. *Inform. Comput.*, 213:13–22, 2012.
- [67] P. Weiner. Linear pattern matching algorithm. In *Proc. 14th Annual IEEE Symposium on Switching and Automata Theory*, pages 1–11, Washington, DC, USA, 1973. IEEE.
- [68] Peter Weiner. The file transmission problem. In *Proceedings of the June 4-8, 1973, National Computer Conference and Exposition*, pages 453–453. ACM, 1973.
- [69] Dan E Willard. Log-logarithmic worst-case range queries are possible in space $\Theta(N)$. *Information Processing Letters*, 17(2):81–84, 1983.